On the Link between Structural Dependencies and Software Changes

Gustavo Ansaldi Oliva

THESIS PRESENTED
TO THE
INSTITUTE OF MATHEMATICS AND STATISTICS
OF THE
UNIVERSITY OF SÃO PAULO
FOR
OBTAINING THE TITLE
OF
PH.D. IN COMPUTER SCIENCE

Program: Ph.D. in Computer Science

Supervisor: Marco Aurélio Gerosa, Ph.D.

During the development of this author, the author received financial support from CAPES, CNPq, European Commission, and Hewlett-Packard Brazil

São Paulo, September 2016

On the Link between Structural Dependencies and Software Changes

This is the original version of the thesis elaborated by
the candidate Gustavo Ansaldi Oliva, just as
submitted to the Judging Committee.

Abstract

OLIVA, G. A. **On the Link between Structural Dependencies and Software Changes**. 2016. 78 f. Ph.D. Thesis - Institute of Mathematics and Statistics (IME), University of São Paulo (USP), São Paulo, 2016.

Low structural coupling is a design principle at the heart of software engineering. A recurrent claim is that classes with high coupling are prone to undergo forced local changes as a consequence of changes made in the classes they are connected to. This claim can be found in almost every Software Engineering book, in fundamental papers of the area, in whitepapers written by industry experts, and even in Wikipedia. Despite the popularity and credibility of the claim, very little research effort has been put on its understanding. In other words, if a class A depends on another class B, then is A more likely to co-change with B as compared to the case where A does not depend on B? In other words, is the existence of dependencies statistically associated with the occurrence of co-changes? To what extent? Answering this question is crucial step in understanding the link between dependencies and software changes. Hence, in this paper, we set out to empirically investigate the link between structural dependencies and co-changes. In a preliminary study with 4 open-source systems, we discovered that structural dependencies do not instantly make artifacts co-change. However, the rate with which an artifact co-cochanges with another is indeed higher when the former structurally depends on the latter. We confirmed this finding in a new study where we extracted structural dependencies and co-changes from 77,286 code snapshots of 45 Java projects randomly sampled from the Apache Software Foundation. Our results indicated that, when A depends on B and B changes, the chances of A changing together with B is around 32% in average, with a standard deviation of 13.6%. We also built classification models using Random Forests to investigate which kinds of dependency best explain co-changes. We found that the length of transitive dependencies, number of type imports, number of method calls, and number of references were the most important variables in the model. However, the classifiers were often inaccurate, thus implying that dependencies are not good predictors for co-changes. In fact, we also found that a substantial number of commits involve classes that are not connected via dependencies, reinforcing our belief that co-changes are more frequently induced by other forms of connascence, such as conceptual coupling. In summary, on the one hand, we found empirical evidence connecting the existence of structural dependencies to the occurrence co-changes. On the other hand, we found that the majority of co-changes do not correlate with structural dependencies, meaning that structural dependencies might be responsible for a small portion of all software changes. As a practical consequence, developers should still embrace the low coupling principle by managing software dependencies while designing and evolving their systems. However, our findings also imply that IDEs should take into account additional sources of information to support developers in successfully propagating software changes, as structural

dependencies seem not to be the main player. Finally, the data and tools produced during this research might be leveraged to bootstrap follow-up investigations, such as the influence of structural anti-patterns on change propagation.

Keywords: structural dependencies, syntactic dependencies, co-changes, change propagation, change coupling, evolutionary coupling, static analysis, historical analysis, classification models, random forests.

Resumo

OLIVA, G. A. **Sobre a Conexão entre Dependências Estruturais e Mudanças no Software.** 2016. 78 f. Tese (Doutorado) - Instituto de Matemática e Estatística (IME), Universidade de São Paulo (USP), São Paulo, 2016.

Baixo acoplamento estrutural é um princípio de design que está no coração da engenharia de software. Um discurso recorrente é de que classes com alto acoplamento são mais propensas a sofrerem mudanças forçadas locais por consequência de mudanças realizadas nas classes as quais estão conectadas. Essa asserção pode ser encontrada em quase todos os livros de Engenharia de Software, em artifos fundamentais da área, em whitepapers escritos por especialistas da indústria e até na Wikipedia. Apesar da popularidade e credibilidade do princípio, pouquíssimo esforço de pesquisa foi colocado na direção de seu entendimento. Em outras palavras, se uma classe A depende de outra classe B, então A é mais propenso a mudar conjuntamente com B comparado à situação em que A não depende de B? Ou seja, a existência de dependências é estatisticamente associada com a ocorrência de mudanças casadas? Até que ponto? Responder essa questão é um passo crucial para um entendimento da conexão entre dependências estruturais e mudanças casadas. Em um estudo preliminar com 4 sistemas de código aberto, descobrimos que dependências estruturais não fazem com que artefatos fatalmente mudem de forma casada. Contudo, a taxa com a qual um artefato muda conjuntamente com outro é, de fato, maior quando o primeiro estruturalmente depende do segundo. Nós confirmamos essa descoberta em um novo estudo em que extraímos dependências estruturais e mudanças casadas de 77.286 snapshots de código de 45 projetos Java aleatoriamente selecionados da população de projetos da Apache Software Foundation. Nossos resultados indicaram que, quando A depende de B e B muda, a chance de A mudar junto com B é de 32% em média, com um desvio padrão de 13.6%. Também construímos modelos de classificação usando Florestas Aleatórias para investigar quais tipos de dependência melhor explicam mudanças casadas. Descobrimos que o comprimento formado do caminho formado por dependências transitivas, o número de importações de tipe, o número de chamadas de métodos e o números de referências foram as variáveis mais importantes no modelo. Contudo, os classificadores foram frequentemente imprecisos, implicando assim que dependências não são bons preditores para mudanças casadas. De fato, também concluímos que um número substancial de commits envolvem classes que não estão conectadas por dependências, reforçando nossa crença de que mudanças casadas são mas frequentemente induzidas por outras formas de conascença, tais como acoplamento conceitual. Em resumo, por um lado encontramos evidência empírica conectando a existência de dependências estruturais com a ocorrência de mudanças casadas. Por outro lado, descobrimos que uma quantidade substancial de mudanças casadas não está correlacionada com dependências estruturais, implicando que essas dependências são provavelmente responsáveis por uma porção de pequena de todas as mudanças

no software. Como consequência prática, desenvolvedores devem continuar a abraçar o princípio do baixo acoplamento por meio da gerência de dependências durante o design e evolução de seus sistemas. Contudo, os resultados também implicam que IDEs devem passar a considerar adicionais fontes de informação para dar suporte ao desenvolvedores na tarefa de propagação das mudanças, já que dependências estruturais não parecem ser o ator principal nesse cenário. Finalmente, os dados e ferramentas produzidos durante essa pesquisa podem ser aproveitados para alavancar investigações seguintes, tal como a influência de anti-padrões estruturais em propagação de mudanças.

Palavras-chave: dependências estruturais, dependências sintáticas, mudanças casadas, propagação de mudanças, acoplamento de mudança, acoplamento evolucionário, análise estática, análise histórica, modelos de classificação, florestas aleatórias.

Contents

List of Acronyms and Abbreviations

ASF	Apache Software Foundation
CVS	Concurrent Version System
IDE	Integrated Development Environment
MSR	Mining Software Repositories
UML	Unified Modeling Language
SVN	Subversion
VCS	Version Control System

List of Figures

List of Tables

Chapter 1

Introduction

Since the notorious software crisis from the late 60's (Naur and Randell, 1969), practitioners and researchers have sought better ways to develop software systems. During this quest, some fundamental Software Engineering development principles were conceived. Larry Constantine introduced a key principle: *design systems with 'low coupling'* (Constantine, 1968). Coupling is the manner and degree of interdependence between software modules[1]. Constantine and colleagues claimed that "minimizing connection between modules also minimizes the paths along which changes and errors can propagate into other parts of the system, thus eliminating disastrous ripple effects" (Stevens et al., 1974).

The low coupling principle is still highly regarded as of today. The common sense still is that, in tightly coupled systems, "a change in one module usually forces a ripple effect of changes in other modules" (Wikipedia, 2016). Similar statements can be found in several Software Engineering books:

"A class with high (or strong) coupling relies on many other classes. Such classes may be undesirable; some suffer from the following problems: Forced local changes because of changes in related classes. (...) Low Coupling supports the design of classes that are more independent, which reduces the impact of change" (Larman, 2004)

"In software design, we strive for lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect" [STE74], caused when errors occur at one location and propagate through a system." (Pressman, 2009)

"Classes that are tightly coupled are hard to reuse in isolation, since they depend on each other. Tight coupling leads to monolithic systems, where you can't change or remove a class without understanding and changing many other classes. The system becomes a dense mass that's hard to learn, port, and maintain." (Gamma et al., 1994)

"Not only outgoing dependencies cause trouble, but also incoming ones. This design disharmony means that a change in an operation implies many (small) changes to a lot of different operations and classes" (Lanza and Marinescu, 2006)

In academia, researchers have also extensively discussed the principle (Geipel and Schweitzer, 2012; Hassan and Holt, 2004; Sangal et al., 2005; Stevens et al., 1974; Wirth, 1971). In industry, recognized experts have also long discussed the relationship between software dependencies and coupling, including Robert Martin (2006) and Martin Fowler (2001).

[1] ISO/IEC/IEEE 24765:2010 Systems and software engineering – Vocabulary

Despite the popularity and credibility of the principle, little effort has been put on understanding and quantifying the link between dependencies and change propagation. If a class A depends on another class B, then is A more likely to co-change with B as compared to the case where A does not depend on B? In other words, is the existence of dependencies associated with the occurrence of co-changes? How frequently? The answer to these questions will shed light into how much effort developers should put on minimizing structural coupling between classes or modules, which is a far from trivial task. In addition, discovering whether specific kinds of dependencies are more likely to induce change propagation might help developers even more. The other side of the coin is also utterly important. How many changes are motived by structural dependencies? Can dependencies be used to predict co-changes? The will highlight whether IDEs should take into account other sources of information besides structural relationships to support developers in successfully propagating software changes, as incomplete or incorrect changes are known to lead to bugs (Hassan and Holt, 2004; Zimmermann et al., 2005). This functionality is particularly important to newcomers, who have alleged they often have problems finding the correct artifacts to address an open issue (Steinmacher et al., 2016).

1.1 Problem Statement

Minimizing change propagation is a desirable software quality, because it eases software maintenance. Structural dependencies have long been blamed to act as paths through which changes propagate. Developers are taught to manage dependencies while designing, developing, and evolving their systems. However, the extent to which dependencies indeed propagate changes is unknown. Whether specific dependencies are more likely to propagate changes is also unknown. This leaves developers rather clueless as to how much effort should be spent into managing dependencies in order to minimize change propagation.

Thesis statement: *Although dependencies increase the likelihood that a client will be affected by changes to the supplier, most software changes scatter throughout the system via paths that do not correspond to structural dependencies, meaning that change propagation cannot be significantly minimized or appropriately dealt with by following these dependencies.*

Our findings revealed that, in average, a co-change is 20% more likely to occur when a class A depends on another class B as compared to when A does not depend on B. Even though the low coupling principle holds in general, this rate differs from project to project. Therefore, in this thesis we employed a mix of historical and analyses to acquire a deeper understanding of the link between dependencies and software changes, as well as to glean actionable information that might help developers or tool builders.

1.2 Major Thesis Contributions

This thesis demonstrates that:

- When a class A depends on another class B and B changes, the likelihood that A will change together with B is 32% in average, being around 20% higher in average than the likelihood found in the case where A does not depend on B. Hence, the "low coupling" principle holds.

- In the majority of cases, our classification models showed that no two kinds of dependencies are redundant, meaning that there is no silver bullet: all kinds of structural dependencies contribute to explaining co-changes. This requires a fairly sophisticated dependency extraction tool.
- Despite the preprocessing and powerful classification algorithm, our classifiers were often inaccurate, implying that structural dependencies are bad predictors for co-changes. In other orders, it is very likely that most co-changes occur because of factors that are not directly associated with structural dependencies.

As additional contributions, we emphasize the tools we developed and the data made available. The toolset can be used to further explore the link between dependencies and co-changes. The datasets, which track the evolution of the dependencies networks of all studied systems, can also be leveraged to conduct different sorts of evolutionary analyses.

1.3 Publications

1. `[Conference Paper]` *Experience Report: How do Structural Dependencies Influence Change Propagation? An Empirical Study* (Chapter 4)

   Gustavo A. Oliva and Marco A. Gerosa. In Proceedings of the 26th IEEE International Symposium on Software Reliability (ISSRE), pages 250-260, Gaithersburg, MD, USA, 2015. IEEE Computer Society Press. (Acceptance ratio: 55/172 = 32%, Qualis A2 [2]).

2. `[Journal Paper]` *On the Link between Structural Dependencies and Software Changes* (Chapter 5)

   Gustavo A. Oliva, Christoph Treude, Marco A. Gerosa. To be submitted to the Empirical Software Engineering journal.

1.3.1 Supporting Publications

1. `[Book Chapter]` *Change Coupling between Software Artifacts: Learning from Past Changes*

   Gustavo A. Oliva and Marco A. Gerosa. The Art and Science of Analyzing Software Data, ed. Christian Bird, Tim Menzies, and Thomas Zimmermann, pages 285-324. ISBN 978-0124115194. Morgan Kaufmann, 1st edition, 2015.

2. `[Workshop Paper]` *What Can Commit Metadata Tell us About Design Degradation?*

   Gustavo A. Oliva, Igor Steinmacher, Igor Wiese, Marco A. Gerosa. In Proceedings of the 2013 International Workshop on Principles of Software Evolution (IWPSE), pages 18-27, Saint Petersburg, Russia, 2013. ACM. (Acceptance ratio: 10/21 = 48%)

3. `[Workshop Paper]` *IVAR: A Conceptual Framework for Dependency Management*

   Gustavo A. Oliva, Marco A. Gerosa. IX Workshop de Manutenção de Software Moderna (WMSWM), Fortaleza, Brazil, 2012.

[2] Based on the report issued by Capes in 2012: https://www.capes.gov.br/images/stories/download/avaliacao/Comunicado_004_2012_Ciencia_da_Computacao.pdf

4. [Workshop Paper] *Preprocessing Change-Sets to Improve Logical Dependencies Identification*

   Gustavo A. Oliva, Francisco W. Santana, Cleidson R. B. de Souza, Marco A. Gerosa. In Proceedings of the 6th International Workshop on Software Quality and Maintainability (SQM), pages 17-24, Szeged, Hungary, 2012. Software Improvement Group (SIG). (Acceptance ratio: 7/16 = 44%)

5. [Doctoral Symposium Paper] *A Method for the Identification of Logical Dependencies*

   Gustavo A. Oliva, Marco A. Gerosa. In Proceedings of the 7th International Conference on Global Software Engineering (ICGSE) Workshops, pages 70-72, Porto Alegre, Brazil, 2012. IEEE Computer Society Press.

1.3.2 Parallel Researches

1. [Conference Paper] *A Change Impact Analysis Approach for Workflow Repository Management*

   Gustavo A. Oliva, Marco A. Gerosa, Dejan Milojicic, Virginia Smith. In Proceedings of the 20th IEEE International Conference on Web Services (ICWS) - Applications and Experience Track, pages 308-315, Santa Clara, CA, USA, 2013. (Qualis A1[2], *Invited for Special Issue*)

2. [Journal Paper] *A Static Change Impact Analysis Approach based on Metrics and Visualizations to Support the Evolution of Workflow Repositories*

   Gustavo A. Oliva, Marco A. Gerosa, Fabio Kon, Dejan Milojicic, and Virginia Smith. International Journal of Web Services Research (IJWSR) - Special Issue on Data Quality in Big Data and Trust (volume 13, issue 2), pages 74-101, 2016. ISSN: 1545-7362. IGI Global. (Invited extension of "A Change Impact Analysis Approach for Workflow Repository Management", Impact factor: 0.257 [3])

3. [Conference Paper] *Characterizing Key Developers: A Case Study with Apache Ant*

   Gustavo A. Oliva, Francisco W. Santana, Kleverton C. M. de Oliveira, Cleidson R. B. de Souza, and Marco A. Gerosa. In Proceedings of the 18th International Conference on Collaboration and Technology (CRIWG), pages 97-112, Raesfeld, Germany, 2012. Springer Berlin Heidelberg. (Qualis B1[2], *Invited for Special Issue*)

4. [Journal Paper] *Evolving the System's Core: A Case Study on the Identification and Characterization of Key Developers in Apache Ant*

   Gustavo A. Oliva, José Teodoro da Silva, Marco A. Gerosa, Francisco W. Santana, Claudia M. L. Werner, Cleidson R. B. de Souza, and Kleverton C. M. de Oliveira. Computing and Informatics (CAI) - Special Issue on Selected Papers from CRIWG 2012 (volume 34, issue 3), pages 678-724, 2015. ISSN: 1335-9150. Slovak Academy of Sciences. (Invited extension of "Characterizing Key Developers: A Case Study with Apache Ant", Impact factor: 0.524 [3])

[3] Based on the 2015 InCities™ Journal Citation Reports®, Thomson Reuters

1.3.3 Co-authorship

1. [Journal Paper] *Using contextual information to predict co-changes*

   Igor S. Wiese, Reginaldo Ré, Igor Steinmacher, Rodrigo T. Kuroda, Gustavo A. Oliva, Christoph Treude, Marco A. Gerosa. Accepted for publication in Journal of Systems and Software (JSS) - Special Issue on , 2016. ISSN: 0164-1212. Elsevier. (Impact factor: 1.424 [3])

2. [Conference Paper] *An Empirical Study of the Relation Between Strong Change Coupling and Defects Using History and Social Metrics in the Apache Aries Project*

   Igor S. Wiese, Rodrigo T. Kuroda, Reginaldo Ré, Gustavo A. Oliva, and Marco A. Gerosa. In Proceedings of the 11th IFIP WG 2.13 International Conference on Open Source Systems (OSS), pages 3-12, Florence, Italy, 2015. Springer International Publishing. (Qualis B2[2])

3. [Workshop Paper] *Are the methods in your data access objects (DAOs) in the right place? A preliminary study*

   Maurício F. Aniche, Gustavo A. Oliva, Marco A. Gerosa. In Proceedings of the 6th IEEE International Workshop on Managing Technical Debt (MTD), pages 47-50, Victoria, BC, Canada, 2014. CPS Conference Publishing Services.

4. [Conference Paper] *Using Structural Holes Metrics from Communication Networks to Predict Change Dependencies*

   Igor S. Wiese, Rodrigo T. Kuroda, Douglas N. R. Junior, Reginaldo Ré, Gustavo A. Oliva, Marco A. Gerosa. In Proceedings of the 20th International Conference on Collaboration and Technology (CRIWG), pages 294-310, Santiago, Chile, 2014. Springer International Publishing. (Qualis B1[2])

5. [Conference Paper] *What do the Asserts in a Unit Test Tell us about Code Quality? A Study on Open Source and Industrial Projects*

   Maurício F. Aniche, Gustavo A. Oliva, Marco A. Gerosa. In Proceedings of the 17th European Conference on Software Maintenance and Evolution (CSMR), pages 111-120, Genova, Italy, 2013. IEEE Computer Society Press. (Qualis A2[2])

6. [Workshop Paper] *Understanding Complex Software Ecosystems: The Role of Core-Periphery Identification*

   Francisco W. Santana, Gustavo A. Oliva, Marco A. Gerosa, Cleidson R. B. de Souza. The Future of Collaborative Software Development (FutureCSD 2012)

7. [Book Chapter] *An Integrated Development and Runtime Environment for the Future Internet*

   Amira B. Hamida, Fabio Kon, Gustavo A. Oliva, Carlos E. M. dos Santos, Jean-Pierre Lorré, Marco Autili, Guglielmo de Angelis, Apostolo Zarras, Nikolaos Georgantas, Valérie Issarny, and Antonia Bertolino. The Future Internet, 81-92.

8. [Journal Paper] *A Systematic Literature Review of Service Choreography Adaptation*

   Leonardo A. F. Leite, Gustavo A. Oliva, Guilherme M. Nogueira, Marco A. Gerosa, Fabio Kon, and Dejan Milojicic. Service Oriented Computing and Applications 7 (3), 199-216.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. In Chapter 2, we provide a brief background on software changes and structural dependencies. In Chapter 3, we present related research to our analysis on the relationship between structural dependencies and software changes. In Chapter 4, we present a warm-up study in which investigated the link between dependencies and co-changes in 4 open-source Java systems. In Chapter 5, we present our complete study on the topic, where we investigated 45 open-source Java systems from the Apache Software Foundation. In Chapter 6, we state our conclusions and plans for future work.

Chapter 2

Background and Definitions

2.1 Software Changes

To keep end-users satisfied, developers change software systems to introduce new feature and fix bugs. To accomplish either task, it is common for developers to simultaneously modify more than one artifacts (e.g., class) of the system. But why do artifacts co-change? In the early 90s, Page-Jones tried to answer this question by introducing the concept of "connascence" (Page-Jones, 1992). The term connascence is derived from Latin and means "having been born together." The Free Dictionary defines connascence as: (a) the common birth of two or more at the same time, (b) that which is born or produced with another, and (c) the act of growing together. Page-Jones borrowed the term and adapted it to the software engineering context: "connascence exists when two software elements must be changed together in some circumstance in order to preserve software correctness" (Page-Jones, 1999).

Connascence assumes several forms and can be either explicit or implicit. To illustrate this point, consider the following code excerpt1 written in Java and assume that its first line represents a software element A and that its second line represents a software element B:

```
String s; //Element A (single source code line)
s = "some string"; //Element B (single source code line)
```

There are (at least) two examples of connascence involving elements `A` and `B`. If `A` is changed to `int s;` then `B` will have to be changed too. This is called type connascence. Instead, if `A` is changed to `String str;` then `B` will need to be changed to `str = "some string"`. This is called name connascence. These two forms of connascence are called explicit. A popular manifestation of explicit connascence comes in the form of structural dependencies (e.g., methods calls), which is the target of this thesis. In turn, as we mentioned before, connascence can also be implicit, such as when a certain class needs to provide some functionality described in a design document.

In general, connascence involving two elements A and B occurs because of two distinct situations:

- A depends on B, B depends on A, or both: a classic scenario is when A changes because A structurally depends on B and B is changed, i.e., the change propagates from B to A via a structural dependency (e.g., a method from A calls a method from B). However, this dependency relationship can be less obvious, as in the case where A changes because it structurally depends on B and B structurally depends on C (transitive dependencies). Another

less obvious scenario is when A changes because it semantically depends on B.

- Both A and B depend on something else: this occurs when A and B have pieces of code with similar functionality (e.g., use the same algorithm) and changing B requires changing A to preserve software correctness. As in the previous case, this can be less obvious. For instance, it can be that A belongs to the presentation layer, B belongs to the infrastructure layer, and both have to change to accommodate a new change (e.g., new requirement) and preserve correctness. In this case, A and B depend on the requirement.

Therefore, artifacts often co-change because of connascence relationships. This is what makes artifacts "logically" connected. Most importantly, the theoretical foundation provided by connascence is a key element that justifies the relevance and usefulness of change couplings, as we shall see in the next section.

2.1.1 Change Coupling

Version control systems store and manage the history and current state of source code and documentation. As early as 1997, Ball and colleagues wrote a paper entitled "If your version control system could talk..." , in which they observed that these repositories store a great deal of contextual information about software changes (Ball et al., 1997). Over the years, researchers have leveraged such information to understand how software systems evolve over time, enabling predictions about their properties.

While mining these repositories, researchers observed an interesting pattern: certain artifacts are frequently committed together. These artifacts are connected to each other from an evolutionary point of view, in the sense that their histories intertwine. We call this connection change coupling. We also say that an artifact A is change coupled to B if A often co-changes with B. Other names employed in the literature include logical dependencies/coupling, evolutionary dependencies/coupling, and historical dependencies.

Change coupling can be calculated at different abstraction levels. In this thesis, we focus on file-level change coupling. Analyzing change couplings at this level has two key benefits (D'Ambros et al., 2009): first and foremost, it can reveal hidden relationships that are not present in the code itself or in the documentation. For instance, a certain class A might be change coupled to another class B without structurally depending on it. Second, it relies on historical file co-change information only, which can be easily extracted from commit logs. Therefore, it does not require parsing code, making it more lightweight than structural analysis. It is also programming language agnostic, making it flexible and a good candidate to be used in studies that involve many subject systems written in different languages.

2.2 Software Dependencies

According to the Merriam-Webster dictionary, a dependency refers to "the quality or state of being dependent; especially: the quality or state of being influenced or determined by or subject to another." Software systems comprise artifacts that depend on one another during design time and runtime. These dependencies connecting software artifacts are vaguely known as software dependencies.

2.2.1 Structural Dependencies

Structural dependencies, also known as syntactic dependencies, are a particular kind of software dependency. If a class A depends on another class B, then A is called *client* and B is called *supplier*. Thus, structural dependencies are directional relationships. In the context of object-oriented programming languages like C++, C#, and Java, structural dependencies occur whenever a compilation unit depends on another at either compilation or linkage time. A compilation unit is the smallest unit of source code that can be compiled. In the current implementation of the Java platform, the compilation unit is a file (often with the .java extension).

There are several kinds of structural dependencies, as illustrated in Figure 2.1. In the following, we briefly describe these dependencies, since the subtle differences between them are central to our third research question:

Member-to-member dependencies. These are dependencies where both client and supplier are type members. A type corresponds to either a class or an interface (and several types can be defined within a compilation unit). Type members are the attributes and methods defined within the context of a type. We capture two kinds of member-to-member dependencies, namely:

- Access dependency: a type's attribute access within the body of a certain method.
- Method call dependency: a method invocation (including constructor call) inside a method's body or while defining an attribute.

Member-to-type dependencies. These are dependencies where the client is a method and the supplier is a type, namely:

- Parameter dependency: an input parameter of some type as part of the method signature definition.
- Reference dependency: a reference to a type while defining an attribute or a reference to a type inside a method's body.
- Return dependency: the return type of a method declaration.
- Throw dependency: the exception types thrown as part of the method declaration.

Type-to-type dependencies. These are dependencies where both client and supplier are types, namely:

- Class inheritance dependency: relationship derived from a class that extends another.
- Interface inheritance dependency: relationship derived from an interface that extends another.
- Interface implementation dependency: relationship derived from a class that implements an interface.

Import dependencies. These are dependencies where the client is a compilation unit and the supplier can be either a type or a method (in the case of static imports), namely:

- Type import dependency: a type import declaration as part of the compilation unit header.

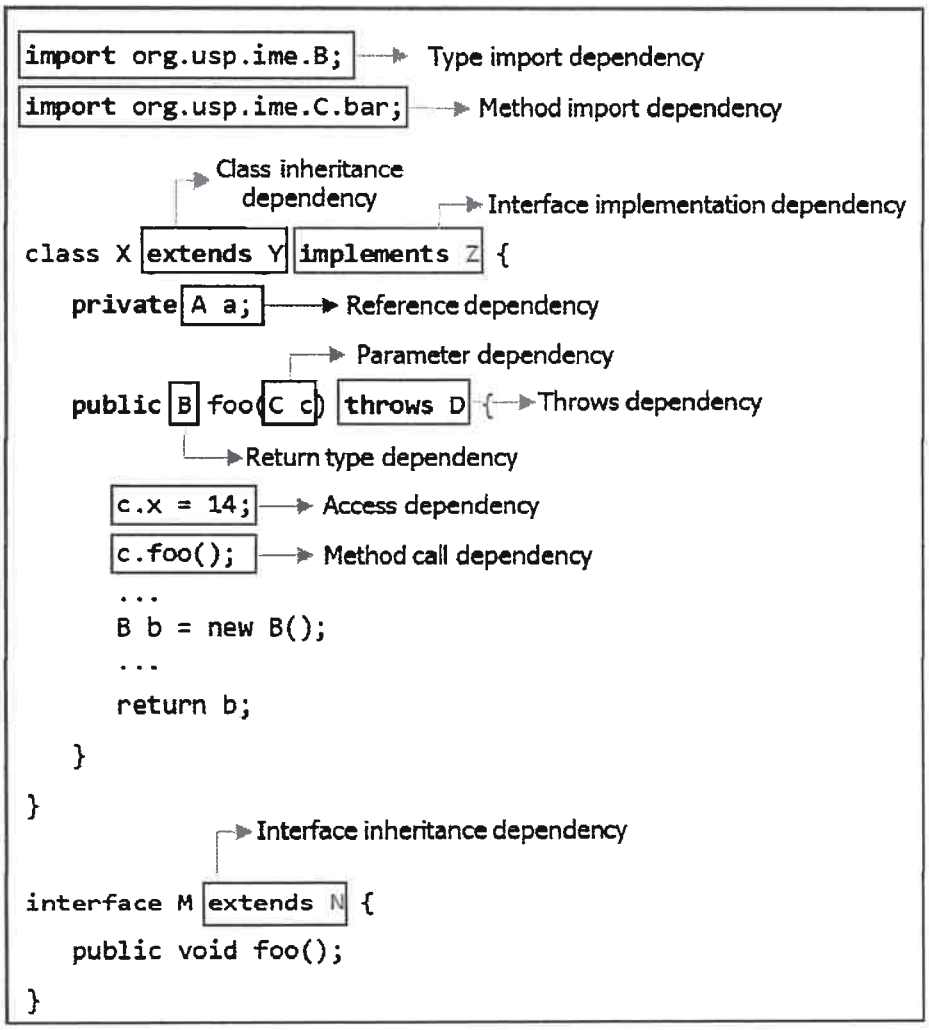

Figure 2.1: *Structural dependencies exemplified in a sample Java code excerpt*

Chapter 3

Related Research

To the best of our knowledge, the work of Geipel and Schweitzer (2012) is the most similar to ours, as they also investigate the link between structural dependencies and co-changes (change propagation) in Java systems. Besides investing a lower number of subject systems (35), they do not track the evolution of the structural dependencies network over time. Instead, they only take into consideration the latest code snapshot when extracting dependencies, arguing that dependencies between two classes i and j are stable from the creation of the younger class until the removal of either i or j. We found that this assumption did not hold for the projects we investigated. Consequently, we recalculate dependencies every time we check out a new version of the code. Finally, all the systems they investigated were hosted in CVS, while ours were all hosted in SVN. Differently from CVS, SVN supports atomic commits, and thus co-changes can be inferred directly from the change-sets. In CVS, change-sets need to be inferred using heuristics, like the sliding time window (Zimmermann and Weißgerber, 2004). Hence, we believe our dataset is more reliable.

Fluri et al. (2005) analyzed the degree to which co-changes are caused by structural changes (which they also call source code coupling) and textual modifications (e.g., software license updates and whitespaces between methods spaces). A preliminary evaluation involving the compare plugin of Eclipse showed that more than 30% of all change transactions did not include any structural change. Therefore, more than 30% of all change transactions have nothing to do with structural coupling. They also found that more than 50% of change transactions had at least one non-structural change. They hypothesize that this could be the result of code ownership/commit habit (a developer works all day in his files and commits everything by the end of the day) and frequent license changes.

Mispropagating changes leads to bugs and increases the cost of developing large software systems. To support developers, Hassan and Holt (2004) studied how change propagation can be predicted. They investigated how a change in one source code entity (function, variable, etc.) propagates to other entities. As opposed to our study, they investigated C systems and only three kinds of relationships: call, use, and define. Later on, Malik and Hassan produced a new version of the study (Malik and Hassan, 2008), in which they described a hybrid an adaptive change propagation predictor that relies on both the software structure and in the development history

3.1 Structural Anti-Patterns

The anti-pattern term was coined in 1995 by Andrew Koenig to describe ineffective or counter-productive patterns (Koenig, 1995). Structural anti-patterns frequently point to design portions

- Method import dependency: a method static import declaration as part of the compilation unit header

More details about the concept of structural dependencies and its applications to software analysis can be found in the systematic review by Arias et al. (2011).

2.3 Java Code Entities

In order to avoid possible misunderstandings due to the plurality of terms and meanings in Software Engineering, we show the terminology we will employ throughout this thesis to refer to Java entities. We focus on the Java terminology because all subject systems evaluated in this study were written in Java. More information can be found at https://docs.oracle.com/javase/tutorial/information/glossary.html

A **compilation unit** is the smallest unit of source code that can be compiled. In the current implementation of the Java platform, the compilation unit is file (often with the .java extension). A **type** is a class or interface. If type X *extends* or *implements* type Y, then X is a **subtype** of Y. We also say that Y is a **supertype** of X. A **class** is a type that defines the implementation of a particular kind of object. A class definition defines instance and class variables and methods, as well as specifying the interfaces the class implements and the immediate superclass of the class. If the superclass is not explicitly specified, the superclass will implicitly be Object. An interface is a collection of method definitions and constant values. An **interface** can be implement by one or more classes. **Type Members** are the attributes and methods defined within the context of a type.

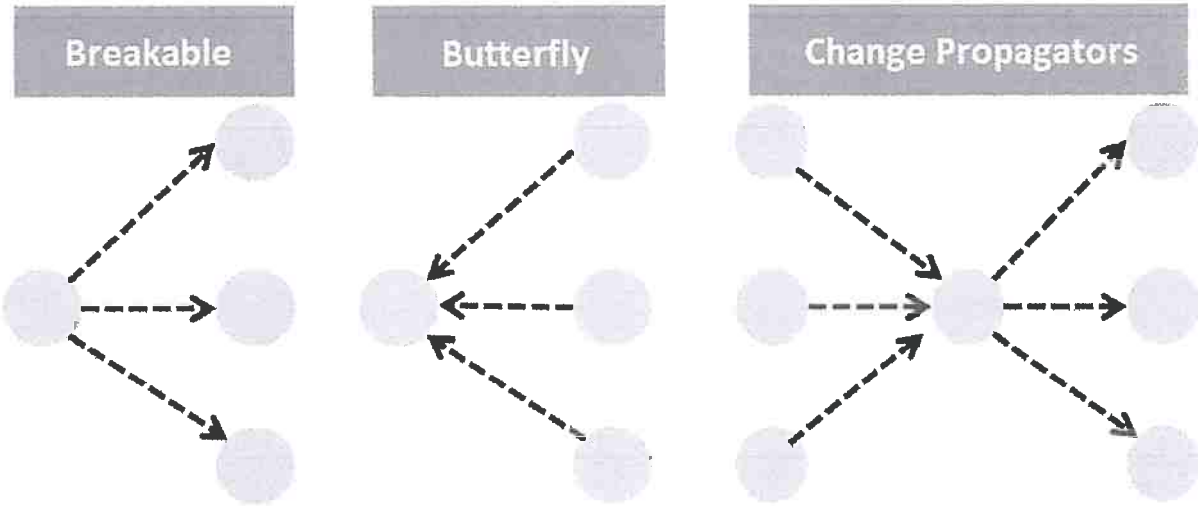

Figure 3.1: *Structural Anti-patterns: Butterfly, Breakable, and Change Propagator*

that should be refactored or treated with special caution. We borrow the terminology employed in IBM SA4J to present some structural anti-patterns (Figure 3.1).

Butterfly. A local butterfly is a compilation unit that has many immediate incoming dependencies. A global butterfly is a compilation unit that has many indirect incoming dependencies and can thus affect many components of the system due to a change. Butterflies are not harmful as long as they comprise stable compilation units (basic interfaces, abstract base classes, etc.).

Breakable. In turn, a local breakable is a compilation unit that has many immediate outgoing dependencies (high coupling). A global breakable is a compilation unit that has many outgoing dependencies and is thus often affected when any other compilation unit in the system is changed.

Hub/Change Propagator. A local hub (a.k.a change propagator) is a compilation unit that is both a butterfly and a breakable at the same time. A global hub is a compilation unit that is often affected when any other compilation unit is changed and also frequently affects a lot of other compilation units when it is changed.

Circular Dependencies and Tangle. A tangle is a strongly connected component of a dependency graph (i.e. every compilation unit is involved in a cycle). An example is shown in Figure 3.2. Cyclic dependencies involving modules are especially harmful and thwart code reuse. Structure 101 and Stan4J provide some support for breaking large tangles

3.2 Design Degradation

The phenomenon of design degradation has been noted since the early days of Software Engineering. Along more than twenty years (1974-1992), Lehman and colleagues developed the laws (or rather empirical hypotheses) of software evolution (Lehman et al., 1997), being among the first ones specifically concerned with continuing change and increasing complexity. In parallel, inspired by the Aristotelian subtance theory, Brooks wrote about the essential and accidental properties of software systems (Brooks, 1987). Brooks also pinpointed complexity and changeability as inherent (essential) properties of software systems. In 1992, inspired by the second law of thermodynamics, Jacobson coined the term software entropy to refer to the increases in software disorder (entropy) over time (Jacobson, 1992). In 1994, Parnas introduced the idea of software aging, by arguing that programs

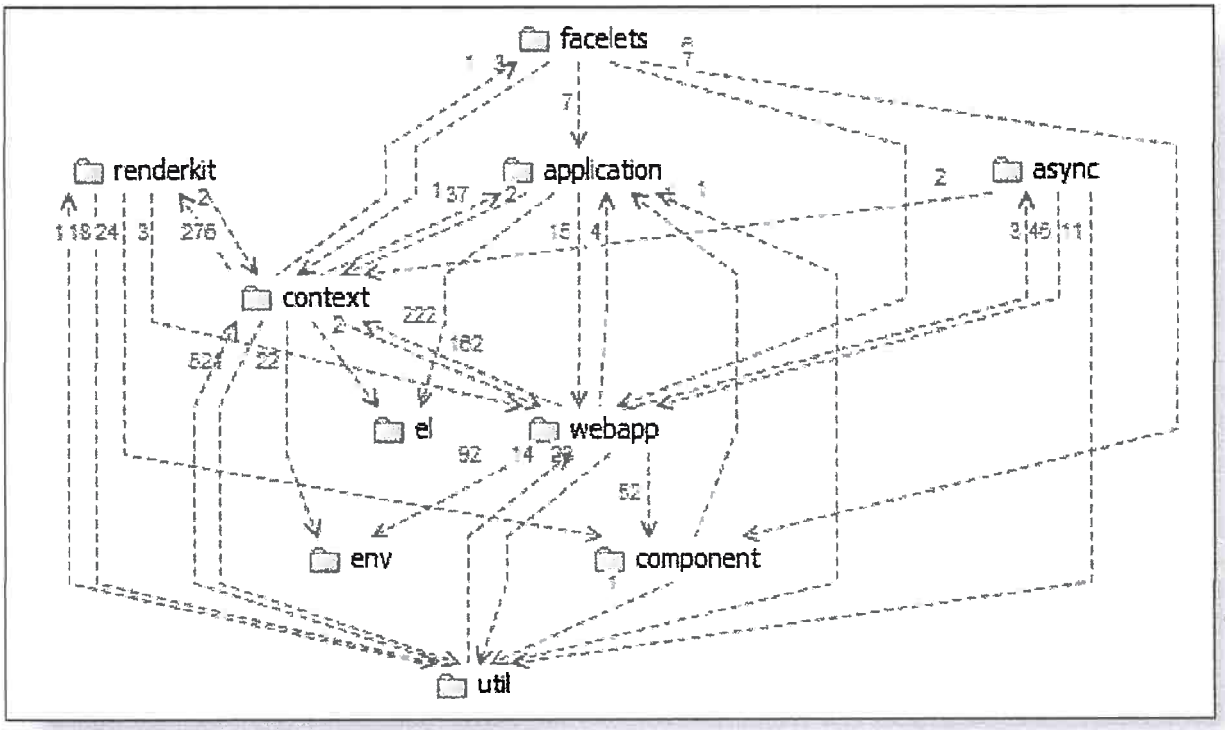

Figure 3.2: *Structural Anti-patterns: Tangles*

get old, just like people (Parnas, 1994).

In early 2000's, Martin introduced a set of design degradation symptoms, which he called "Symptoms of Rotting Design" (Martin, 2000). In the following, we present two of these symptoms:

Rigidity. The definition of rigidity given by Martin is as follows. *Rigidity is the tendency for software to be difficult to change, even in simple ways. A design is rigid if a single change causes a cascade of subsequent changes in dependent modules. The more modules that must be changed, the more rigid the design is. Most developers have faced this situation in one way or another.* (Martin and Martin, 2006). Martin summarizes it as *a design that is difficult to change.*

This anti-pattern has also been studied under the name of ripple effect. An early work on software ripple effect is that of Yau and colleagues, who presented a maintenance framework to cope with program modifications (Yau et al., 1978). Wilkie and Kitchenham (2000) investigated whether classes with high CBO (Coupling Between Objects) metric values are more likely to be affected by change ripple effects. Similarly, Briand and colleagues investigated the use of coupling measures and derived decision models for identifying classes likely to suffer from ripple effect (Briand et al., 1999). Interestingly, these two last studies revealed that highly structurally coupled classes did not always cause significant ripple effects. Conceptual coupling metrics, which are calculated based on semantic information obtained from identifiers and comments in source code, have also been employed to detect ripple effect (Kagdi et al., 2010).

Fragility. The definition of fragility given by Martin is as follows. *Fragility is the tendency of a program to break in many different places when a single change is made. Often, the new problems are in areas that have no conceptual relationship with the area that was changed. Fixing those problems leads to even more problems, and the development team begins to resemble a dog chasing its tail* (Martin and Martin, 2006). Martin summarizes it as *a design that is easy to break*

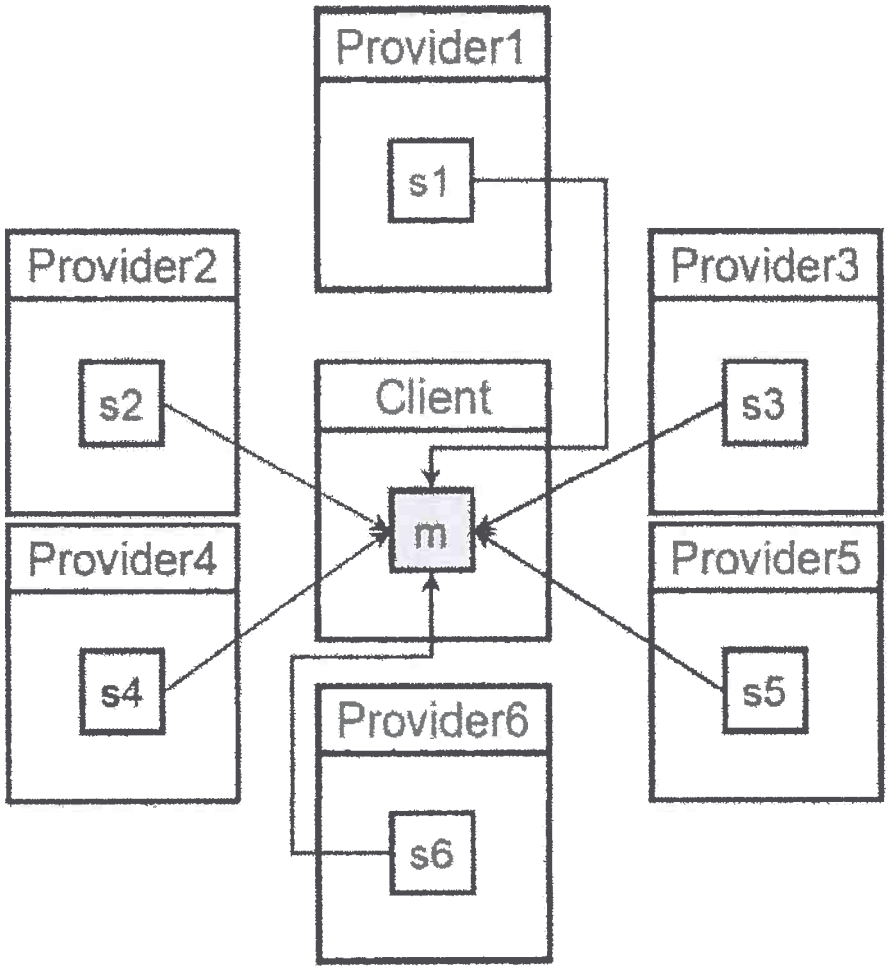

Figure 3.3: *Shotgun Surgery (Lanza and Marinescu, 2006)*

This anti-pattern has also been studied under the name of Shotgun Surgery (Figure 3.3) (Fowler, 1999). We highlight the work of Lanza and colleagues in this area (Lanza and Marinescu, 2006; Marinescu, 2004), who introduced mechanisms called "detection strategies" that combine different code metrics to detect code smells. Lanza and the LOOSE Research Group developed a free tool called iPlasma (Marinescu et al., 2005), which implements the Shotgun Surgery detection strategy (Figure 3.4) by parsing the source code of Java and C# projects. Gîrba et al. (2007) proposed a similar approach to detect this same smell based on the identification of classes that have had their implementation changed together while maintaining their interfaces intact. While all these studies depend on the actual code, our proposed metric relies on commit metadata obtained through the parsing of the log files generated by the version control system. Therefore, the calculation of our metric is fast and does not depend on the programming language in which the software was written.

What kind of changes cause designs to rot? Changes that introduce new and unplanned for dependencies. Each of the four symptoms mentioned above is either directly, or indirectly caused by improper dependencies between the modules of the software. It is the dependency architecture that is degrading, and with it the ability of the software to be maintained.

In a more general context, Gall et al. (2003) mined CVS repositories, collected change couplings and showed that design flaws such as God Classes (Fowler, 1999) and Spaghetti Code (Foote and Yoder, 1999) could be discovered without analyzing the actual source code. D'Ambros and colleagues developed an interactive visualization tool called Evolution Radar (D'Ambros et al.,

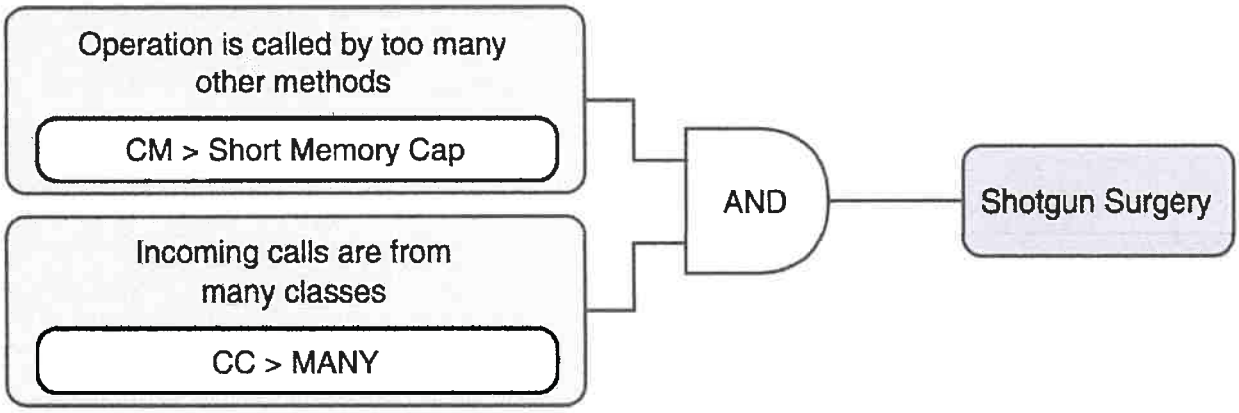

Figure 3.4: *Detection strategy for Shotgun Surgery (Lanza and Marinescu, 2006)*

2009) that displays logical dependencies among modules of a software system. They showed that their tool was able to detect design issues that were not detectable by means of static analysis of code.

3.3 Architectural Conformance Checking

The role played by software architecture in the development process is widely known as crucial. Architectural conformance consists in verifying whether a low-level representation of a software system (e.g. the code) complies with the planned architecture (also known as documented architecture) (Sangal et al., 2005). No conformation means that decisions implemented in code somehow violate the planned architecture, leading to unexpected collateral effects and design degradation. Furthermore, it can be than no conformation means that the planned architecture should be reviewed. Several known techniques and tools aid in architecture conformance. Lattix LDM tool enforces architectural conformance through the specification of architectural rules (in x can-use y and x cannot-use y forms) and partitioning algorithms. Structure 101 tool aids on this task by comparing a user defined architectural diagram to the actual code, resembling the reflexion models technique (Murphy et al., 2001) . JDepend tool accomplishes this task by providing an appropriate Java API that can be used in unit tests.

Chapter 4

Warm-up Study

In this study, we set out to empirically investigate the influence of structural dependencies on change propagation. We focus on a quantitative analysis of how these dependencies relate to the occurrence of co-changes found in the version history of the subject systems. We performed our analysis on 4 open source Java projects, namely Apache Lucene, Apache Tomcat 7, Megamek, and Apache Commons CSV. We answered the following research questions:

(RQ1) Are dependent files more likely subject to co-change than independent ones?

Given a pair of Java files $\langle f_1, f_2 \rangle$, our results indicated that, even though it is more likely that f_1 and f_2 will not co-change just because f_1 depends on f_2 (i.e., dependencies do not instantly make two files change together), the rate with which f_1 co-changes with f_2 is higher when f_1 structurally depends on f_2 (as compared to when f_1 does not depend on f_2). However, this rate is fairly low, with its median ranging from 13.5% (Commons CSV) to 20% (Lucene).

(RQ2) Are dependent files more likely subject to co-change than independent ones when the change context is taken into account?

In this research question, we take the change context into consideration. In this new scenario, we consider that there is a dependency from f_1 to f_2 only if f_1 depends on "something" that changed in f_2 (e.g., a field definition or a method's body). The results vary substantially compared to those we obtained in RQ1. Given a pair of Java files $\langle f_1, f_2 \rangle$, when f_1 depends on f_2 and f_2 changes, it is more likely that f_1 will co-change with f_2 in Lucene and Tomcat. In Commons CSV, the number of co-changes and absence of co-changes is roughly the same in the presence of structural dependencies. In Megamek, it is more likely f_1 will not co-change with f_2.

Furthermore, we also found that the rate with which f_1 co-changes with f_2 is again higher when f_1 structurally depends on f_2 (as compared to when f_1 does not depend on f_2). This rate is also substantially higher than that we found when ignoring the change context (RQ1), ranging from 43.08% (Commons CSV) to 100% (Lucene and Tomcat).

(RQ3) Do changes propagate via structural dependencies?

We focused on an analysis of the proportion of changed methods that propagated changes via call dependencies to other methods per commit. The distribution of this proportion was

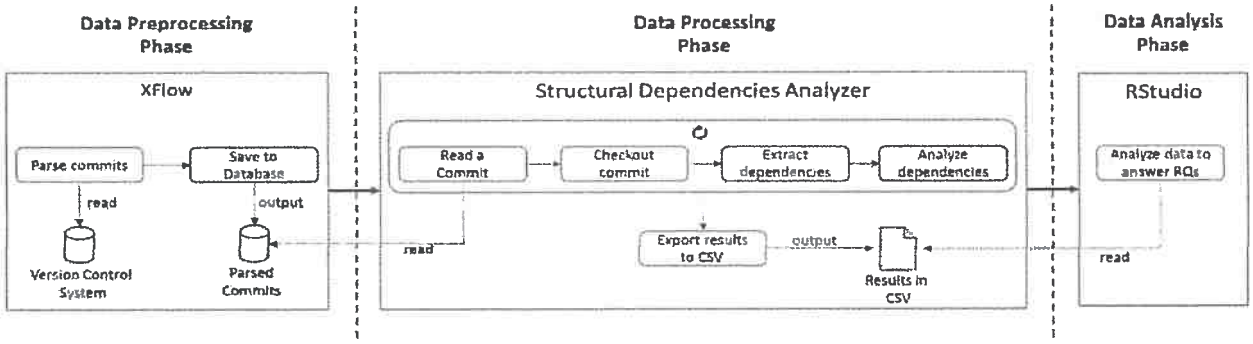

Figure 4.1: *Overview of study design*

similar in all projects, with a median of zero. The mean varied from 11.40% (Tomcat) to 18.55% (Lucene). Hence, we concluded that changes in methods rarely propagate via call dependencies. This result supports the idea found in some related work that few changes are "justified" by the software architecture (Geipel and Schweitzer, 2012).

The key take-home message of this study is that the relationship between structural dependencies and software changes are not as straightforward as one might think. The results of RQ1 and RQ2 reiterate the importance of managing structural dependencies while evolving software systems. This calls for advanced supporting tools that are capable of, for example, measuring the levels of change propagation over time and alerting end-users when thresholds are exceeded. Commercial tools used in industry like IBM RSA and Stan4J only act on a code snapshot and thus are not able to offer features like this. In turn, the results of RQ3 show that the relationship between individual software changes made in a commit might very loosely related to the software architecture. This reinforces the need of researching the relationship between change propagation and the many forms of connascence (Page-Jones, 1992), such as conceptual coupling (Poshyvanyk and Marcus, 2006).

Study organization. The rest of this study is organized as follows. In Section 4.1, we describe the study design, which includes our goals, the procedures for subject systems selection and data collection, and the tools we used. In Section 4.2, we present the detailed results for the research questions. In Section 4.3, we discuss the threats to the validity of this study. Finally, in Section 4.4, we provide a summary of our findings.

4.1 Study Design

In this section, we describe the study design. Figure 4.1 shows an overview of the steps we followed.

4.1.1 Goal

The goal of this study is to empirically determine the influence of structural dependencies on change propagation. The focus is on the evolution of open-source Java applications. Our perspective is that of developers, as they need to be aware of and manage coupling while evolving software systems

4.1.2 Selection of Subject Systems

We pre-established a set of requirements for choosing the subject projects. First, as the identification of structural dependencies is programming language dependent, we decided to pick projects written in the same language. We chose Java, since there are many open0source projects written in this language and we had an appropriate tool to extract Java dependencies. This decision also enabled us to compare the results across projects. Second, due to limitations of our tool set, we chose projects versioned with Subversion (SVN) only. Third, in order to improve the external validity of our findings, we selected projects of different sizes and domains. Finally, since we want to investigate the link between structural dependencies and co-changes, we need to extract these dependencies in every code snapshot of the subject system. In this warm-up study, we limited our analysis to 4 systems.

4.1.3 Data Collection

All Apache Software Foundation projects in Subversion are hosted under the same repository. As of August 24, 2015, this repository hosts more than 1.69 million commits. Working with large remote repositories poses a series of challenges. To cope with them, we built a local mirror of the whole repository. In fact, we already had this mirror in hands, as it was built for previous studies we conducted (Oliva and Gerosa, 2011).

4.1.4 Data Preprocessing

We used XFlow to collect, parse, and store commit metadata (Figure 4.1 – Data Preprocessing Phase). We relied on such data to determine which commits to analyze, as we are only interested in commits that included Java files. We also relied on these data to gain more insight about the subject systems, including their age, number of developers, and number of Java files per commit.

4.1.5 Data Processing and Analysis

In this section, we detail the data processing and analysis phases of the study design. As depicted in Figure 4.1, for each parsed commit, we read it, check-out the corresponding code snapshot, extract dependencies, and analyze them according to the specific research question at hand. After all code snapshots are analyzed, we export the results to a CSV file and analyze it using RStudio. More details about the tools we used in this study are given in Section 4.1.6.

In the following, we discuss the dependency analysis method we employed to answer each of three research questions we raised in this study.

a) RQ1 - Data Processing and Analysis

To address this research question, we mined the version control system to capture dependencies and co-changes (algorithm I describes the steps we followed in details). For each commit, the corresponding code is checked-out. Afterwards, each file f_2 in the commit is compared with each file f_1 in the checked-out code in order to discover two things: whether f_1 is a client of f_2 (line 07) and whether they co-changed (line 08). A file f_1 is a client of f_2 if there is at least one dependency from f_1 to f_2 (e.g., when a certain method from f_1 calls a method defined in f_2). The kinds of

Table 4.1: *Relationship Status*

File pair $\langle f_1, f_2 \rangle$		**Does f_1 depend on f_2?**	
		Yes (there is at least one dep. from f_1 to f_2	No (there are no deps. from f_1 to f_2)
Does f_1 co-change with f_2?	Yes (f_2 changes and f_1 changes as well)	CD	$C\overline{D}$
	No (f_2 changes and f_1 *does not* change)	$\overline{C}D$	$\overline{CD}$

dependencies we capture are described in Section 2.2.1. We also say that a file f_1 co-changes with f_2 when both are included in the same commit.

A *rel* (relationship) object stores how many times f_1 depended (and did not depend) on f_2, as well as how many times f_1 co-changed (and did not co-change) with f_2. This object is kept in memory and is retrieved on demand (line 05). The rel object for f_1 and f_2 is updated every time f_2 is included in a commit and f_1 is in the checked-out code.

Algorithm I: Recording co-changes and structural dependencies

```
01. for each commit c do
02.   code <- checkout(c)
03.   for each file $f_2$ in commit c do
04.     for each file $f_1 \neq f_2$ in the code do
05.       rel <- getRelationship($f_1$,$f_2$)
06.       if(rel is null) createRel($f_1$,$f_2$)
07.       isClient <- is $f_1$ a client of $f_2$?
08.       hasCoChanged <- does c contain $f_1$?
09.       rel.update(isClient,hasCoChanged)
```

After mining all commits, we obtained the data shown in Table 4.1 for every single pair of files evaluated. From such data, we derived a series of metrics:

- $CoChanges(f_1, f_2)$: Number of commits in which f_1 co-changed with f_2: $CD + C\overline{D}$
- $NoCoChanges(f_1, f_2)$: Number of commits in which f_1 did not co-change with f_2: $\overline{C}D + \overline{CD}$
- $Dep(f_1, f_2)$: Number of commits that included f_2 and f_1 depended on f_2: $CD + \overline{C}D$
- $NoDep(f_1, f_2)$: Number of commits that included f_2 and f_1 did not depend on f_2: $C\overline{D} + \overline{CD}$
- $CoChangeRatioWithDep(f_1, f_2)$: Of all commits that included f_2 and f_1 depended on f_2, how many times did f_1 co-change with f_2? $CD/(CD + \overline{C}D) = CD/Dep(f_1, f_2)$
- $CoChangeRatioWithoutDep(f_1, f_2)$: Of all commits that included f_2 and f_1 did not depend on f_2, how many times did f_1 co-change with f_2? $C\overline{D}/(C\overline{D} + \overline{CD}) = C\overline{D}/NoDep(f_1, f_2)$

These metrics were exported to a CSV file and then statistically analyzed in RStudio (Figure 4.1). As a preliminary analysis, we compared the distributions of CD and $\overline{CD}$ to understand the frequency of co-changes in the presence of structural dependencies. Next, to answer the research question, we investigated whether files with structural dependencies are more likely to co-change. We accomplished that by checking whether the rate with which f_1 co-changes with f_2 is higher when f_1 structurally depends on f_2 (as compared to when f_1 does not depend on f_2). That is, we expect CoChangeRatioWithDep(f_1,f_2) to be higher than CoChangeRatioWithoutDep(f_1,f_2) in most cases. This intuition was formalized as an experiment with the following hypotheses:

NullHypothesis(H_0): Occurrence of co-changes involving f_1 and f_2 is not higher when f_1 depends on f_2.

AlternativeHypothesis(H_1): Occurrence of co-changes involving f_1 and f_2 is higher when f_1 depends on f_2.

Our findings for this research question are presented in Section 4.2.2.

b) RQ2 - Data Processing and Analysis

To answer this research question, it becomes imperative to be able to identify the different kinds of changes. In the scope of this study, we focus on investigating the extent to which the three following kinds of change propagate:

(i) Added methods and changed methods: in a commit, a developer might add methods to a class and change existing methods. For instance, the method foo() from the class A might be modified to call the method bar() from class B. To the purpose of this analysis, constructors and the set of lines corresponding to the declaration of attributes are both considered methods.

(ii) Added types: in a commit, a developer might add a new type to either a new class or an existing class.

(iii) Types with changed set of type-to-type (t2t) dependencies: in a commit, a developer might add or remove a type dependency for a certain type s/he is working on. For instance, a developer might make a certain class A extend a class B or make a class C implement an interface D instead of an interface E.

The following algorithm describes the approach we employed to detect added and changed methods (i).

Algorithm II: Capturing added and changed methods

```
01. for each commit c do
02.   code <- checkout(c)
03.   prevCode <- checkout(previous(c))
04.   for each file f in commit c do
05.     methods <- getMethods(f,code)
06.     prevMethods <- getMethods(f,prevCode)
07.     methodComp <- MethodComparator.run(methods,prevMethods)
08.     record(c,methodComp)
```

Subroutine: MethodComparator.run(methods,prevMethods)

```
01.  identical <- removeIdentical(methods,prevMethods)
02.  changed <- removeChanged(methods,prevMethods)
03.  deleted <- prevMethods
04.  added <- methods
05.  methodComp <- new MethodComp(identical,changed,deleted,added)
06.  return methodComp
```

We start by getting the list of methods from a certain artifact in the current code snapshot and in the previous code snapshot (line 01-06). We then compare the two lists of methods (line 07) and record the changes (line 08). We detect added and removed methods using a simple syntactic heuristic. First, identical methods are removed from both lists (subroutine, line 01). We then capture changed methods (subroutine, line 02). We consider that a method m changed into a method m' when their signatures are the same, their return type is the same and their body is different. We then remove these changed methods from both lists. The remaining methods in the prevMethods list are deemed as deleted (subroutine, line 03). The remaining ones in the methods list are deemed as added.

With relation to finding the added types (ii), we proceed in a similar fashion of Algorithm II. Let the types found in the previous version of a file be prevTypes and let the types found in the current version of a file be types. We deem two types as identical when they have the exact same source code. Let identicalTypes be the set with the identical types. The set of added types correspond to types minus identicalTypes.

Finally, in order to detect changes in type-to-type dependencies (iii), we employed Algorithm III.

Algorithm III: Capturing types with changed set of type-to-type (t2t) dependencies

```
01. for each commit c do
02.   code <- checkout(c)
03.   prevCode <- checkout(previous(c))
04.   for each file f in commit c do
05.     changedTypes <- empty list
06.     types <- getTypes(f,code)
07.     prevTypes <- getTypes(f,prevCode)
08.     pairs<prevType,type) <- types with same FQN, but different code
09.     for each pair in pairs do
10.       if (pair.prevType.getT2TDeps $\neq$ (pair.type.getT2TDeps)
11.         changedTypes.add(type)
12.     record(c,changedTypes)
```

We first get the list of types from a certain artifact in the current code snapshot and in the previous code snapshot (line 01-07). We then discover the types with same FQN but with different source code (line 08). Next, we examine these (pairs of) types to discover the cases where type-to-type dependencies were added or removed (line 09-11). Finally, we record the results (line 12).

Now that we showed how the three kinds of changes we defined were detected, we can rely on Algorithm I to answer the research question. However, the way the client is determined changes now (line 07). Instead of checking whether there is any dependency from f_1 to f_2, we check the low-level software changes (i), (ii), and (iii) we previously defined. More specifically, we perform the steps described in Algorithm IV.

Algorithm IV: Deciding whether there is a dependency from f_1 to f_2 while considering the change context

```
01.  deps <- JDX.getAllDeps($f_1$,$f_2$)
02.  for each dep do
03.    addedM <- get added methods in commit
04.    removedM <- get removed methods in commit
05.    if (addedM or changedM contains dep.Supp) do
06.      isClient <- true
07.      break
08.    addedT <- get added types in commit
09.    changedT <- get types with changed t2t deps
10.    if (addedT or changedT contains dep.Supp) do
11.      isClient <- true
12.      break
13.  return isClient
```

The main idea is to discover whether f_1 has a dependency to something that changed in f_2. Hence, we first get all kinds of dependencies from f_1 to f_2 (line 01). Next, we check whether the supplier of any of these dependencies is either an added or changed method (lines 03-07) or an added or changed type (lines 08-12). If we find at least a single positive case, then we state that f_1 is a client of f_2.

Finally, just as we did for RQ1, we check whether the rate with which f_1 co-changes with f_2 is higher when f_1 structurally depends on f_2 (as compared to when f_1 does not depend on f_2). This intuition is again formalized as an experiment with the following hypotheses:

Nullhypothesis(H_0): Occurrence of co-changes involving f_1 and f_2 is not higher when f_1 depends on f_2 and the change context is taken into account.

AlternativeHypothesis(H_1): Occurrence of co-changes involving f_1 and f_2 is higher when f_1 depends on f_2 and the change context is taken into account.

Our findings for this research question are presented in Section 4.2.3.

c) RQ3 - Data Processing and Analysis

To answer this research question, we identified the changes that happened in a commit and inferred whether they propagated via structural dependencies. For the purpose of this analysis, we focused on added and changed methods. The detailed steps we followed are described in Algorithm V.

Algorithm V: Determining change propagation via structural dependencies

```
01. for each commit c do
02.   code <- checkout(c)
03.   addedChangedM <- discover added and changed methods
04.   if (addedChangedM > 0) do
05.     deps <- JDX.getCallDeps(code)
06.     callGraph <- Jung.getGraph(deps)
07.     callGraphT <- Jung.getTransitiveClosure(callGraph)
08.     propagatedM <- 0
09.     for each method m in addedChangedM do
10.       preds <- callGraphT.getPred(m)
11.       for each method pred in preds do
12.         if(m $\neq$ pred and
13.           addedChangedM constains pred) do
14.           distance <- Dijkstra.calcDistance(callGraph,pred,m)
15.           if (distance <= 3) do
16.             propagatedM <- propagated + 1
17.             break
18.     percentProp <- propagated / size(addedChangedM)
19.     record(c,percentProp)
```

For each commit, we check-out the code (line 02) and discover the added and changed methods (line 03) using the approach we described in Section 4.1.5. If we find at least one added or changed method, we proceed with the analysis. Otherwise, we skip the commit and investigate the following one (line 04). Using JDX, we obtain all call dependencies found in the code (line 05). With these dependencies in hands, we use Jung to build a call-graph (line 06). A call-graph is a directed graph where vertices are methods and edges represent calling relationships. More specifically, the edges connect a method m to another method m' if and only if there is at least one call dependency from m to m'. Subsequently, we use Jung once more, but this time to calculate the transitive closure of the call-graph (line 07). The transitive closure is a graph in which there is an edge connecting v to v' if and only if there is a path from v to v' in the original graph. In our case, the transitive closure of the call-graph is a directed graph in which an edge from m to m' implies in the existence of a transitive dependency in the original call-graph. In line 08, we initialize a counting variable that will record the number of added/changed methods that propagated changes via call dependencies. We consider that an added/changed method m propagates a change via call dependencies if at least one of its predecessors in the transitive closure is a method m' that was either added or changed in the commit. However, since an edge in the transitive closure stands for a transitive dependency, we check back in the original call-graph how many edges (hops) were needed to connect m' to m. We thus use the Dijkstra's shortest path algorithm (line 14) to determine the length of the shortest path that connects m' to m in the original call-graph. However, since this path might be quite lengthy, we restrict the maximum distance to 3 (line 15). In other words, we accept only two intermediary methods in the path that connects m' to m. The rationale is the following: if too many hops are needed to connect m' to m, then it is more likely that they changed together because of some other reason that is not directly related to the calling dependencies. For instance, it could be the case that two classes are semantically related, with similar terms present in their comments and identifiers

(Poshyvanyk and Marcus, 2006). Finally, we record the proportion of added/changed methods that propagated changes in the commit (line 19). Once all data are captured, we analyze the distribution using descriptive statistics.

Our findings for this research question are presented in Section 4.2.4.

4.1.6 Supporting Tools

XFlow. This is an extensible, interactive, and stand-alone tool we have developed Santana et al. (2011), whose general goal is to provide a comprehensive software evolution analysis by mining software repositories and taking into account both technical and social aspects of the developed systems. In this study, we employed XFlow to parse and store commit logs. Project website: https://github.com/golivax/xflow2

JDX. Java Dependency eXtractor (JDX) is a core tool in this study. It is a Java library we have developed to extract dependencies from Java code. The tool is robust and is able to extract all dependencies listed in Section 2.2.1. The library relies on the robust Java Development Tools Core (JDT Core) library, which is the incremental compiler used by the Eclipse IDE. As a desirable consequence, JDX is able to handle Java source code in its plain form. Project website: https://github.com/golivax/JDX

Jung. Java Universal Network/Graph Framework (Jung) is a Java library that provides a common and extensible language for modeling, analyzing, and visualizing data that can be represented as a graph or network. In this study, we employed Jung to help us compute certain algorithms, such as the transitive closure of a directed (dependency) graph. Project website: http://jung.sourceforge.net

Structural Dependencies Analyzer. This is a custom tool we built for the purpose of answering the research questions of this study. It relies on JDX and Jung. It implements all algorithms shown in Section 4.1.5.

R and RStudio. All statistical analyses of this study were conducted using the R package v3.1.2., with the support of RStudio IDE v.0.98.1103.

4.2 Results

This section presents and discusses the results of our three research questions. For each research question, before presenting the results, we first briefly recall our motivation to study it and the approach we employed to answer it. All data and scripts used in this study are available at: https://github.com/golivax/issre2015.

4.2.1 Preliminary Analysis: Characterizing Subject Systems

Given the criteria listed in Section 4.1.2, we selected the 4 following projects: Apache Lucene (core), Apache Tomcat 7, Megamek, and Apache Commons CSV. Lucene is an information retrieval library widely used in the implementation of Internet search engines. Tomcat is a renowned web server and servlet container that implements several Java EE specifications, Megamek is a turn-based strategy game inspired by the "Classic BattleTech" board game. Commons CSV is a library to read, write, and manipulate CSV files.

Table 4.2: *Description of Subject Systems*

Project	Commit Interval	VCS URL	Mined Path (regex)	Total Commits (w/ Java files)
Apache Lucene (core)	[149570, 926576]	https://svn.apache.org/repos/asf/	/lucene/java/trunk/src/java/.*?\\.java	1962
Apache Tomcat 7	[540106, 1155255]	https://svn.apache.org/repos/asf/	/tomcat/trunk/java/.*?\\.java	3554
Megamek	[2, 11344]	http://svn.code.sf.net/p/megamek/code	/trunk/megamek/src/megamek/.*?\\.java	7754
Apache Commons CSV	[1299104, 1603967]	https://svn.apache.org/repos/asf/	/commons/proper/csv/trunk/src/main/java/.*?\\.java	415

Table 4.2 shows the commit interval we mined for each system. It also depicts the repository URL, the repository path regex, and the total number of commits that matched the regex. We say that a commit matches the regex when it contains at least one file whose path matches the regex. This implies that commits with no Java files were discarded. We also highlight that the reason why the commit intervals seem so big for the Apache projects is because all projects from the Apache Software Foundation are hosted in a single Subversion server. Therefore, we must take a large interval to find all the commits we want for a single system.

4.2.2 RQ1: Are dependent files more likely subject to co-change than independent ones?

Motivation. Classic Software Engineering literature has long stated that structural coupling should be minimized because every time a certain class changes, all other dependent classes are also likely to change, thus inducing ripple effects. Hence, with this research question, we aim to empirically investigate whether dependent files are more likely subject to co-change than independent ones.

Approach. To address this research question, we mine the version control system. For each commit, we search for co-changes and structural dependencies between pairs of files $<f_1,f_2>$, where f_2 is a file in the commit's change-set and f_1 is a file in the checked-out code. Algorithm I describes the steps we followed in details.

Results. We start by showing the results of the comparison of CD and $\overline{C}D$. For this analysis, we selected only the file pairs where $\text{Dep}(f_1,f_2) \neq 0$, i.e., file pairs where f_1 was a client of f_2 in at least one commit. The results for the subject systems are summarized as bean plots in Figure 4.2. The black horizontal lines denote the median of the distributions. The dotted line represents the median of the two distributions combined.

In all systems, the median of CD (left, black) is smaller than the median of $\overline{C}D$ (right, gray). Moreover, the distribution of $\overline{C}D$ has a much larger tail. This suggests that CD is in general smaller than $\overline{C}D$. To further investigate this, we performed a paired two-sample Wilcoxon test (a.k.a. Mann-Whitney test) with the alternative hypothesis that CD is less than $\overline{C}D$ at a significance level of 0.05. We could reject the null hypothesis in all systems with the following p-values: less than 2.2e-16 (Lucene), less than 2.2e-16 (Megamek), equal to 6.506e-06 (CSV Commons), and less than 2.2e-16 (Tomcat).

In summary, we found a consistent behavior across all systems: given a pair of files $<f_1,f_2>$ where f_1 depends on f_2, the number of co-changes (CD) is, in general, smaller than the number of no co-changes ($\overline{C}D$).

(Result I) *The number of times f_1 co-changes with f_2 is smaller than the number of times f_1 does not co-change with f_2 in the context where f_1 depends on f_2 and f_2 changes.*

Next, we investigate whether dependent files are more likely subject to co-change than independent ones. The $CoChangeRatioWithDep(f_1, f_2)$ metric is undefined when both CD and $\overline{C}D$ are

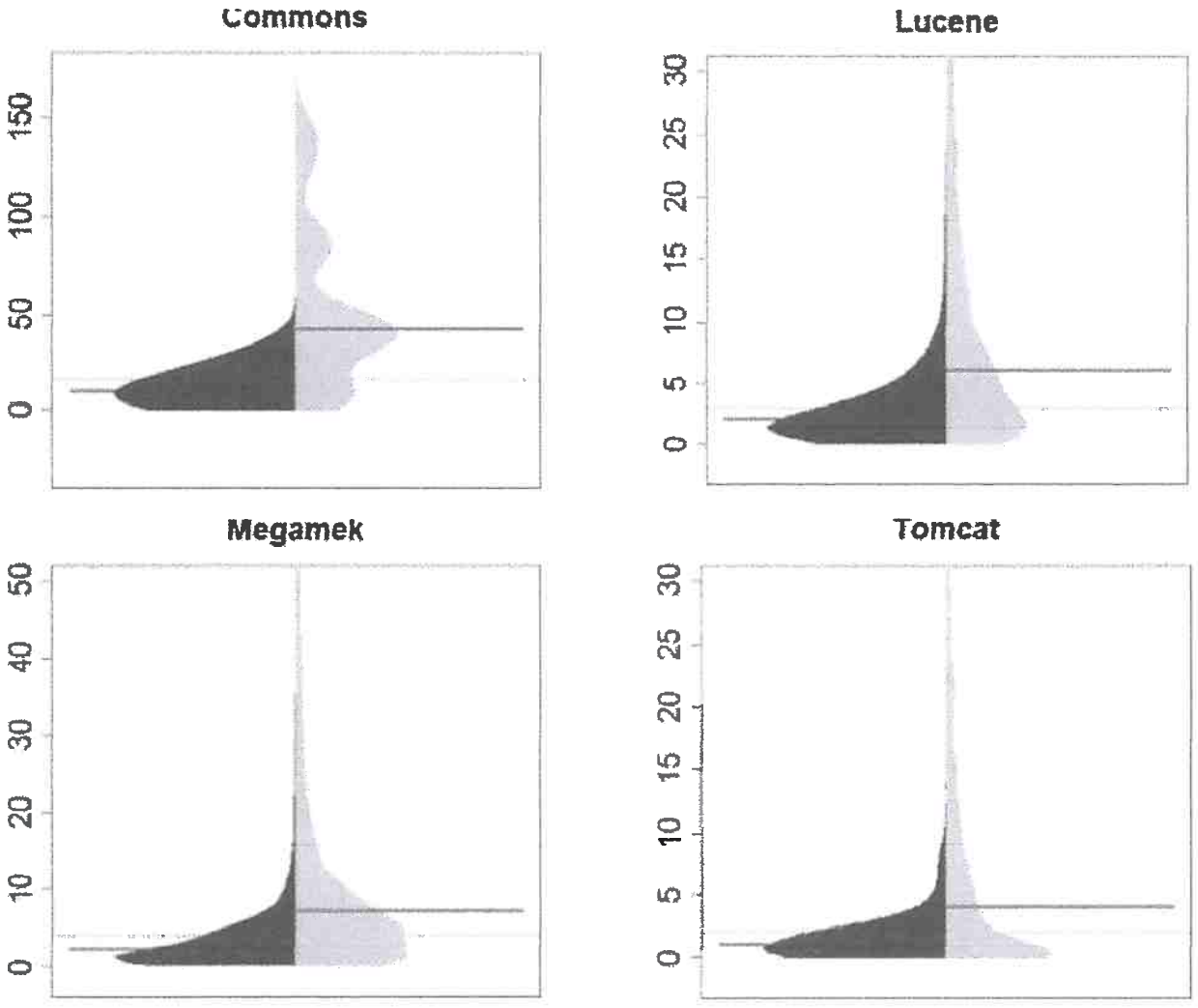

Figure 4.2: *Beanplots comparing the distribution of CD (black) and $\overline{C}D$ gray*

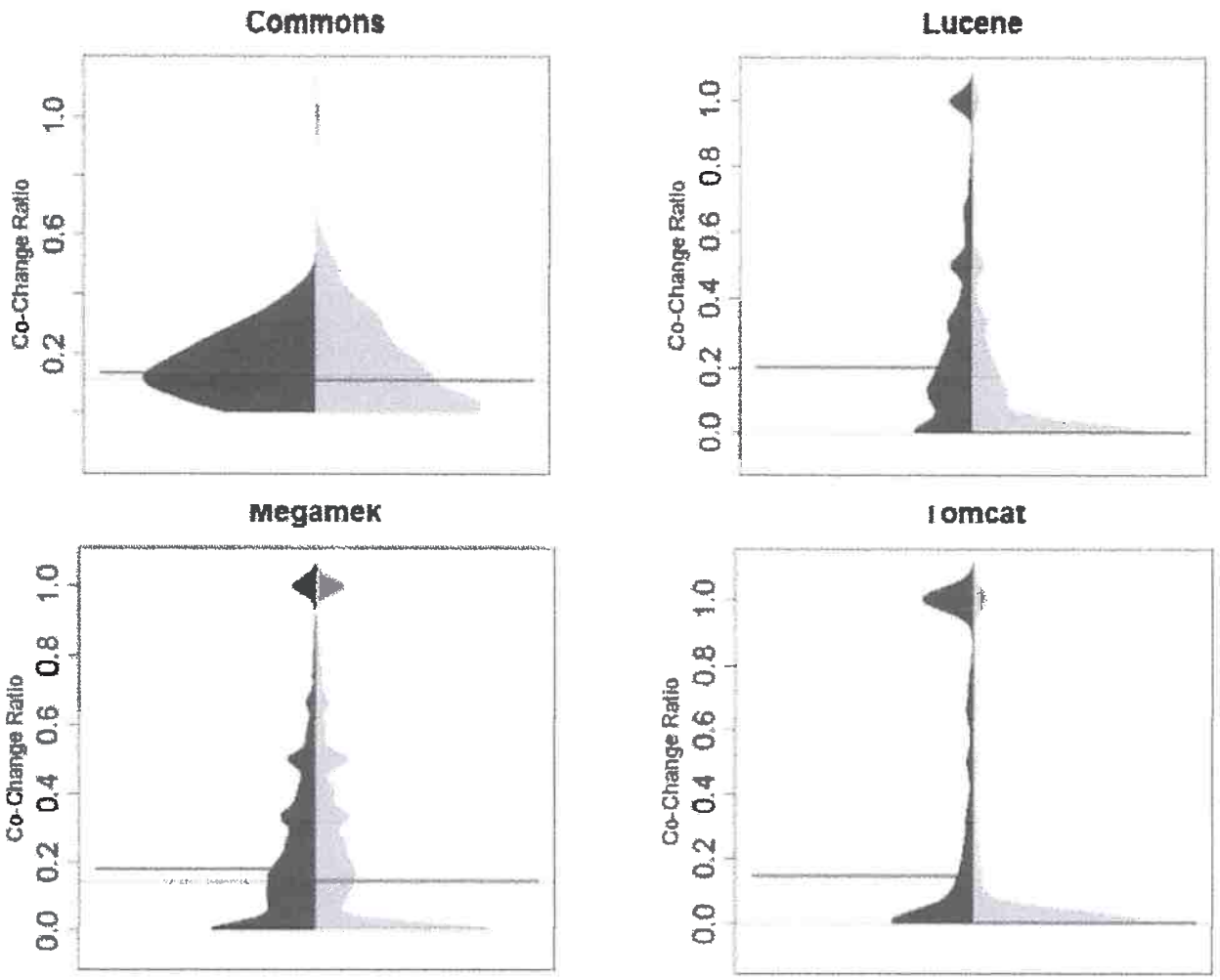

Figure 4.3: *Beanplots comparing the distribution of s_1 (black) and s_2 (gray)*

zero. Analogously, $CoChangeRatioWithoutDep(f_1, f_2)$ is undefined when both $C\overline{D}$ and $\overline{C}D$ are zero. Therefore, for each project, we had to first preprocess the data to remove the cases where these metrics could not be calculated. This led to two samples s_1 and s_2:

- Sample s_1: $CoChangeRatioWithDep(f_1, f_2)$ with $Dep(f_1, f_2) = CD + \overline{C}D > 0$
- Sample s_2: $CoChangeRatioWithoutDep(f_1, f_2)$ with $NoDep(f_1, f_2) = C\overline{D} + \overline{C}D > 0$

As we did in the previous case, we start with a graphical analysis of the distributions of s_1 and s_2 for each project. Figure 4.3 depicts the bean plots we obtained.

In all systems, the median of s_1 is higher than the median of s_2. This supports our hypothesis that the rate with which f_1 co-changes with f_2 is higher when f_1 structurally depends on f_2. However, we highlight that the medians for s_1 are all fairly low: 13.54% for Commons CSV, 20% for Lucene, 17.65% for Megamek, and 14.29% for Tomcat.

The next step was to evaluate whether dependent files are more likely subject to co-change than independent ones (alternative hypothesis). We employed a non-paired two-sample Wilcoxon test at a significance level of 0.05. As opposed to the first test we conducted, this must be non-paired because of the preprocessing we executed. We could reject the null hypothesis for three systems, all of them with a p-value smaller than 2.2e-16. The exception was Apache Commons CSV, since its p-value was equal to 0.1811. In fact, in this system, the medians of s_1 and s_2 were very similar: 0.1354 for

s_1 and 0.1129 for s_2. Hence, in this system, structural dependencies seem not to be associated with co-changes. Given that Commons CSV is a much smaller system (10 classes in v1.2) and it is a library, our hypothesis is that it has very stable interfaces, leading to surgical changes that are not driven by structural relationships.

In summary, in three of the four systems, we could reject the null hypothesis. This provides empirical evidence to support the claim that dependent files are more likely subject to co-change than independent ones.

(Result II) *The rate with which f_1 co-changes with f_2 is higher when f_1 structurally depends on f_2 (as compared to when f_1 does not depend on f_2). However, this rate is fairly low, ranging from 13.5% to 20%.*

As a general conclusion, we can say that even though it is more likely that f_1 and f_2 will not co-change just because f_1 depends on f_2 (i.e., dependencies do not instantly make two files change together), the rate with which f_1 co-changes with f_2 is higher when f_1 structurally depends on f_2.

4.2.3 RQ2: Are dependent files more likely subject to co-change than independent ones when the change context is taken into account?

Motivation. The previous analysis did not take into consideration the specific changes that happened in each file of the commit's change-set. That is, we analyzed dependencies at the file-level. What if we accounted for the specific changes? If a certain method is changed, will its clients (from other classes) change as well? If a certain type now extends another, will the clients of the former be affected?

Approach. The approach we employed is similar to that of RQ1. However, the way the structural client is determined is different. Instead of checking whether there is any dependency from f_1 to f_2 in a given commit, we check whether there is a dependency from f_1 to "something" that changed inside f_2. In the scope of this study, "something" can be either: (i) added/changed methods (e.g., a new method m was added to a type t in f_2), (ii) added types (e.g., a new type t was added to f_2), and type-to-type dependencies (e.g., a type t in f_2 now extends type t' instead of t").

Results. We proceed in the exact same way we did for RQ1. Therefore, we start by showing the results of the comparison of CD and $\overline{C}D$ via bean plots (Figure 4.4). We highlight that we again selected only the file pairs where $\text{Dep}(f_1,f_2) \neq 0$, i.e., file pairs where f_1 was a client of f_2 in at least one commit.

As compared to RQ1, the results are much subtler now, as the medians became much closer to each other in all systems. This is also evidenced by the descriptive statistics for the distributions, which are shown in Table 4.3. Lucene and Tomcat seem to show a similar behavior, with higher number of observations equaling to zero in the right-hand side ($\overline{C}D$) and a higher number of observations equaling to one in the left-hand side (CD). In both systems, the median for CD is one, the median for $\overline{C}D$ is zero, and the mean of CD is higher than the mean of $\overline{C}D$. The shape of the distributions of Commons CSV and Megamek are quite different from the two other systems. In Commons CSV, even though the median of CD is higher than that of $\overline{C}D$, the mean is not. In Megamek, the medians of CD and $\overline{C}D$ are identical, and the mean of $\overline{C}D$ is higher.

To further investigate the data, we performed a series of paired two-sample Wilcoxon tests, all at a significance level of 0.05. For Lucene and Tomcat, we performed the test with the alternative hypothesis that CD *is greater than* $\overline{C}D$. In both cases, we could reject the null hypothesis with a

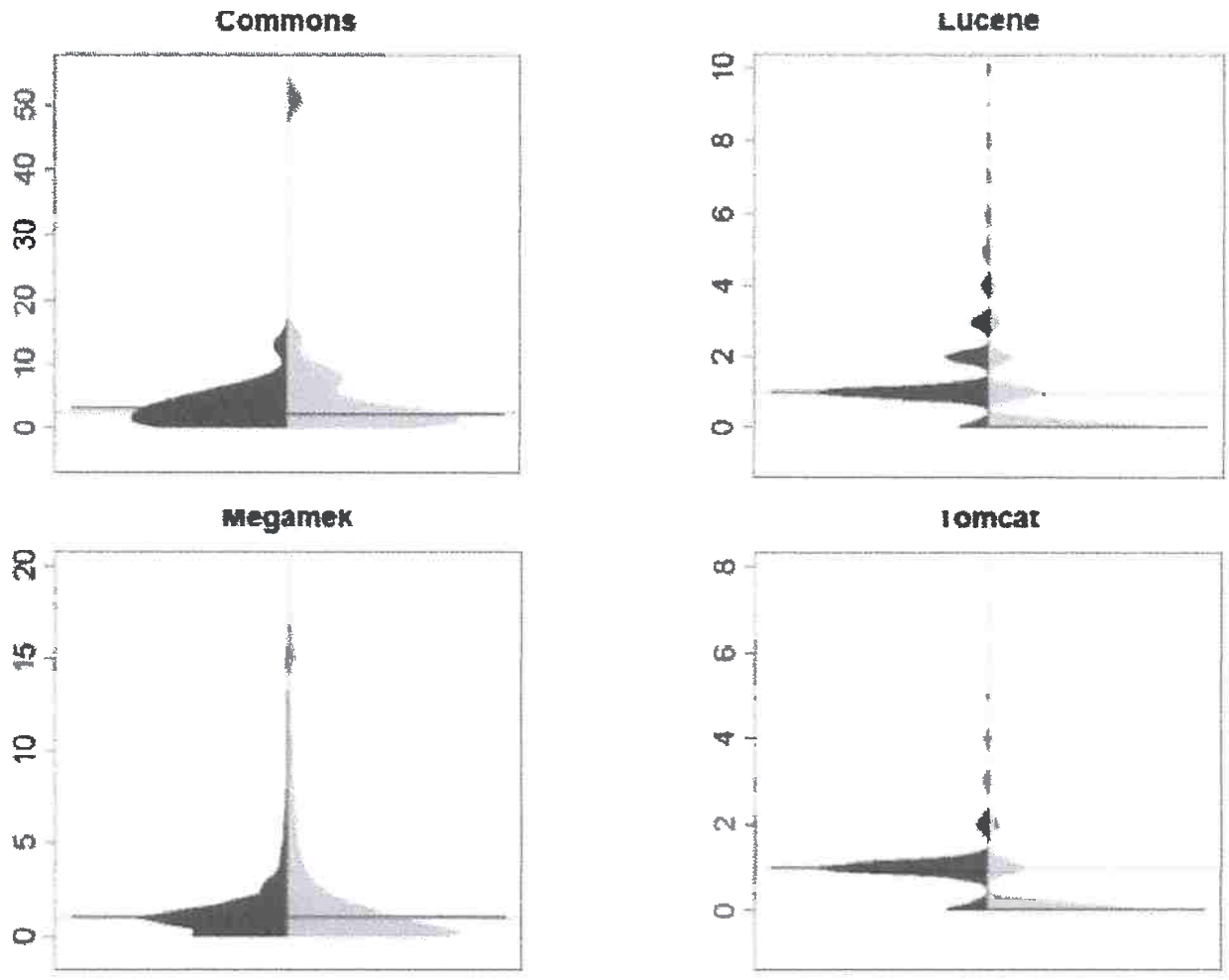

Figure 4.4: *Beanplots comparing the distribution of CD (black) and $\overline{C}D$ gray*

Table 4.3: *Descriptive Statistics for CD and $\overline{C}D$*

System	CD						$\overline{C}D$					
	Min	Q1	Med	Mean	Q3	Max	Min	Q1	Med	Mean	Q3	Max
CommonsCSV	0	1	3	3.04	4	13	0	1	2	5.81	6.75	51
Megamek	0	0	1	1.76	2	305	0	0	1	5.12	3	504
Lucene	0	1	1	1.55	2	31	0	0	0	0.98	1	45
Tomcat	0	1	1	0.97	1	16	0	0	0	0.71	1	38

p-value small than 2.2e-16. For Commons CSV, we tested the alternative hypothesis that CD is less than $\overline{CD}$. The p-value was 0.1562 and thus we could not reject the null hypothesis. In addition, we tested the alternative hypothesis that CD and $\overline{CD}$ are distributions with different shapes (two-sided test). The p-value was 0.3124 and thus we could not reject the null hypothesis again. That is, for Commons CSV, the distributions of CD and $\overline{CD}$ are roughly indistinguishable (as also supported by its bean plot depicted in Figure 4.4). Finally, in Megamek, since the median of CD and $\overline{CD}$ are the same, we first tested the alternative hypothesis that CD and $\overline{CD}$ are distributions with different shapes. We could reject the null hypothesis with a p-value smaller than 2.2e-16., i.e., the location shift is not equal to zero. We then tested whether CD was less than $\overline{CD}$, since the mean of CD (1.76) is smaller than the mean of $\overline{CD}$ (5.12). We could again reject the null hypothesis with the same p-value.

In summary, we compared CD and $\overline{CD}$ to understand the frequency of co-changes in the occurrence of structural dependencies. Given a pair of files $<f_1,f_2>$ with f_1 depending on f_2, the number of co-changes is, in general, greater than the number of no co-changes (absence of co-changes) for Lucene and Tomcat. In Commons CSV, these numbers are roughly the same. In Megamek, the number of co-changes is, in general, smaller than the number of no co-changes. Hence, the results changed substantially as compared to the previous RQ.

(Result III) *When taking the change context into account, the results change substantially as compared to (Result I). In the context where f_1 depends on f_2 and f_2 changes: (a) It is more likely that f_1 will co-change with f_2 (instead of not co-changing) in Lucene and Tomcat; (b) the number of co-changes and no co-changes will be roughly the same in Commons CSV; (c) it is more likely that f_1 will not co-change with f_2 in Megamek*

We now investigate whether dependent files are more likely subject to co-change than independent ones. As we did for RQ1, we check whether the rate with which f_1 co-changes with f_2 is higher when f_1 structurally depends on f_2 (as compared to when f_1 does not depend on f_2). We apply the same preprocessing and obtain the samples s_1 and s_2, which are the target of the analysis. Before evaluating the hypothesis, we start with a graphical analysis of the distributions of s_1 and s_2 for each project. Figure 4.5 depicts the bean plots we obtained.

In all systems, the median of s_1 is significantly higher than the median of s_2. In particular, the median of s_1 is equal to 1.0 in Lucene and Tomcat. This implies that at least 50% of the observations in these distributions are equal to 1, i.e., the client always changed when the supplier changed (the change always propagated). In Megamek, the third quartile of s_2 is equal to 1.0. In turn, the shape of the distributions for Commons CSV looks more uniform, with data more evenly spread in the [0,1] interval. We highlight that the medians for s_1 are significantly higher than those we found in RQ1, as they now ranged from 43.08% (Commons CSV) to 100% (Lucene and Tomcat).

We then evaluated the hypothesis itself using a non-paired two-sample Wilcoxon test at a significance level of 0.05. We could reject the null hypothesis for all systems. Lucene, Megamek, Tomcat had a p-value smaller than 2.2e-16. Commons CSV had a p-value of 0.0015, which is not as strong as the former, but still sufficient to reject the null hypothesis.

In summary, all these results provide empirical evidence to support the claim that dependent files are more likely subject to co-change than independent ones.

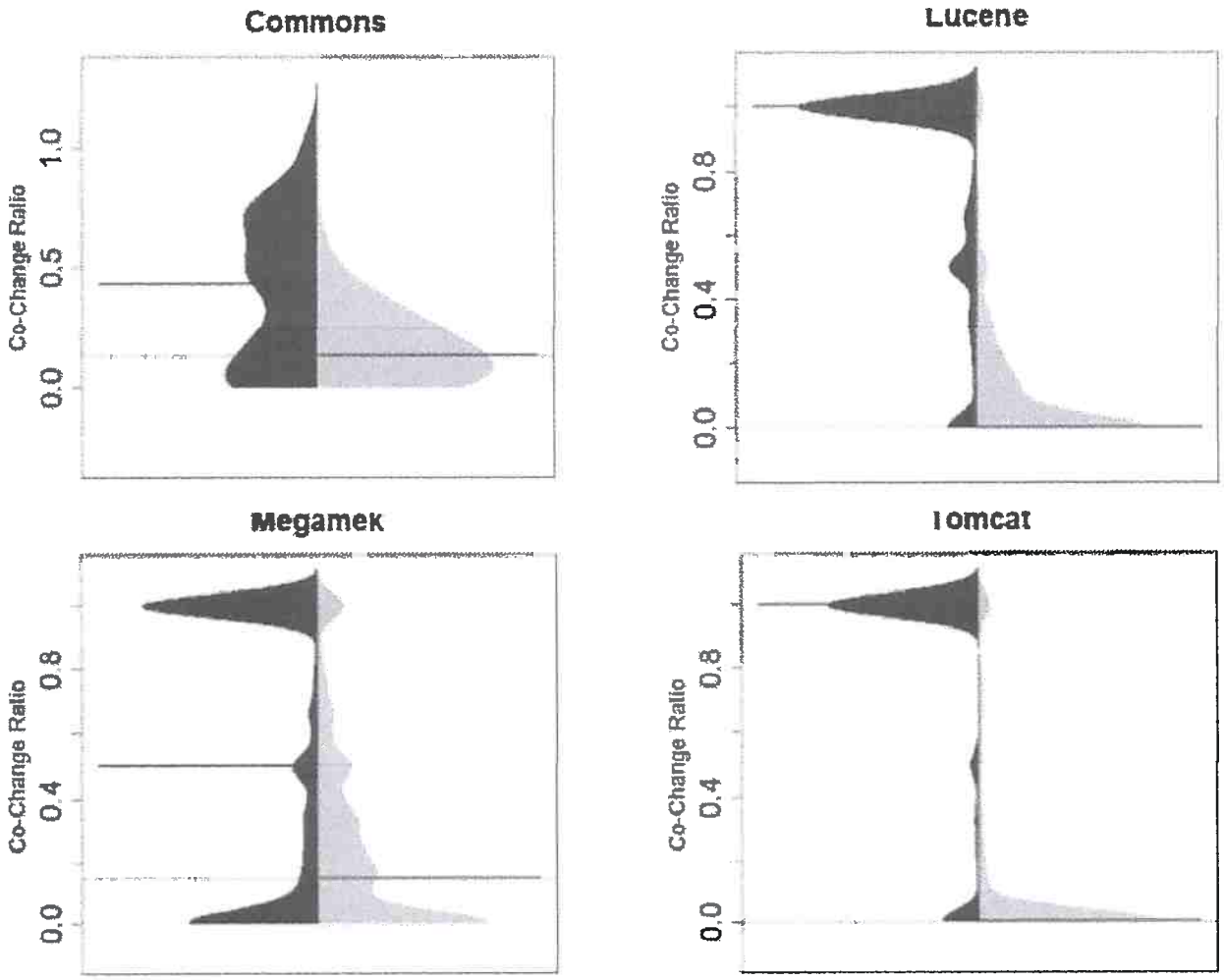

Figure 4.5: *Beanplots comparing the distribution of s_1 (black) and s_2 gray*

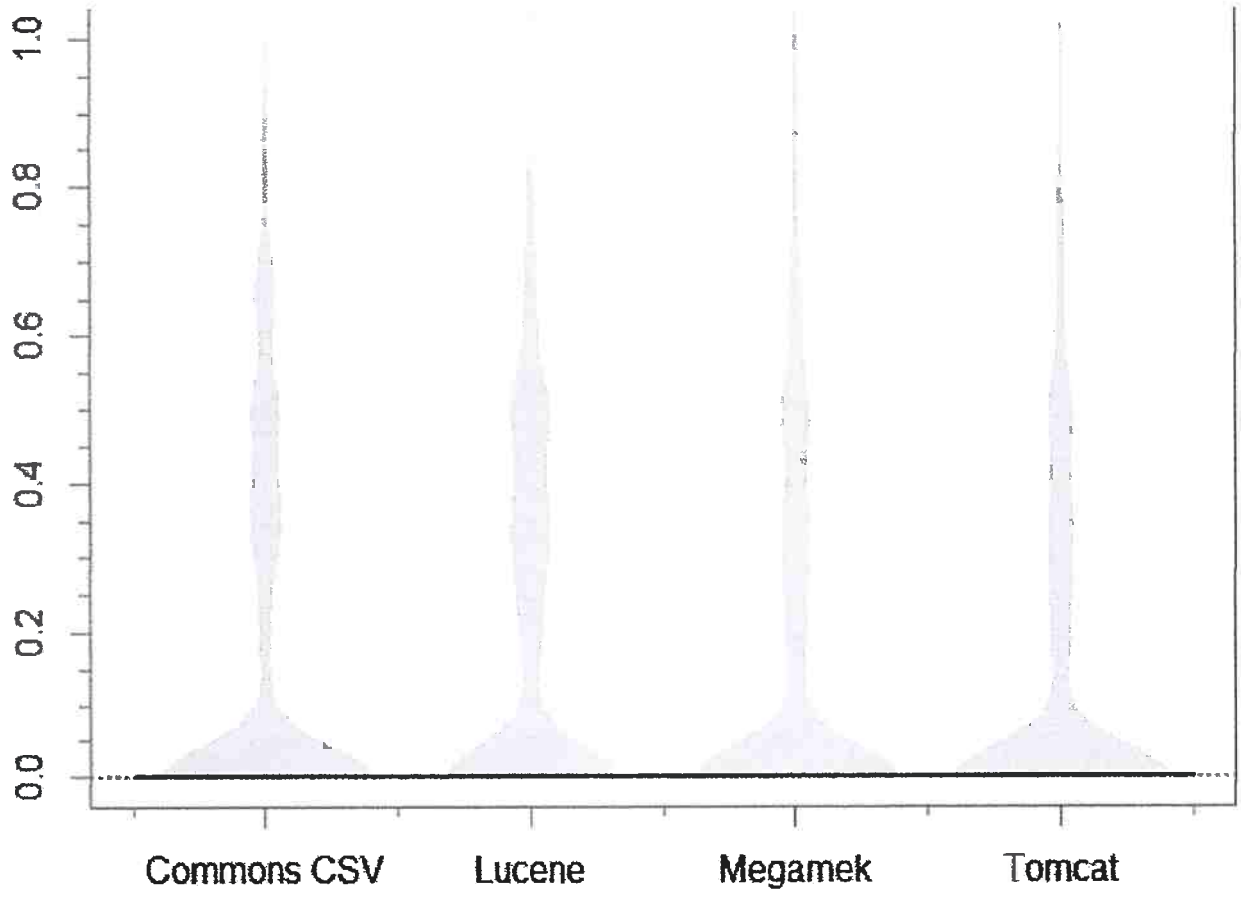

Figure 4.6: *Beanplot for the proportion of methods that propagated changes per commit*

(Result IV) *When taking the change context into consideration, the rate with which f_1 co-changes with f_2 is higher when f_1 structurally depends on f_2 (as compared to when f_1 does not depend on f_2). This rate is also substantially higher than that we found when ignoring the change context (Result II).*

4.2.4 RQ3: Do changes propagate via structural dependencies?

Motivation. Even though structural dependencies might lead to co-changes, it is not clear the extent to which the architecture "justify" software changes (Geipel and Schweitzer, 2012; Zimmermann et al., 2003). How do the actual software changes relate to the architecture? In particular, are there situations where two classes change together and there is no structural connection between them? How often does it happen? These are the aspects we will investigate.

Approach. To answer this research question, we identified the changes that happened in a commit and inferred whether they propagated via structural dependencies using graph algorithms. For the purpose of this analysis, we focused on added and changed methods. The detailed steps we followed are described in Algorithm V.

Results. Figure 4.6 depicts the bean plot for the proportion of added/changed methods that propagated changes via call dependencies per commit. We see a consistent behavior across all systems, as their distributions are very similar. Most importantly, the distributions are very right-skewed and the medians are all zero.

Table 4.4 depicts descriptive statistics for the same variable. The means vary from 11.40%

Table 4.4: *Descriptive Statistics for the Proportion of Methods That Propagated Changes per Commit*

Systems	Min	Q1	Med	Mean	StdDev	Q3	Max
Commons CSV	0	0	0	14.46%	22.77%	33.33%	86.36%
Megamek	0	0	0	13.72%	21.19%	27.27%	100.00%
Lucene	0	0	0	18.55%	22.71%	36.27%	100.00%
Tomcat	0	0	0	11.40%	19.57%	20.00%	88.89%

(Tomcat) to 18.55% (Lucene), and the standard deviations are significant, varying from 19.57% to 22.77%. Hence, the mean is also fairly low for all distributions and there is a significant variability around it.

These results have strong implications in practice: most changes in methods do not tend to propagate via call dependencies. This supports the claim found in the study of Geipel and Schweitzer (2012): very few changes are "justified" by the software architecture.

(Result V) *Changes in methods rarely propagate via call dependencies.*

4.3 Threats to Validity

Some aspects may have pose threats to the validity of this study. First, our toolset has some limitations. Algorithm I relies on paths for distinguishing between files. Hence, it does not account for renames and file moving. Furthermore, in RQ2 we do not deal with removed methods. Although method removals could be the trigger of change propagations chains, we believe that adding and changing methods are more usual operations done by developers. Hence, we believe our results are still reliable, in the sense that they cover enough ground.

In RQ2, we deem "changed types" as those that had their type-to-type dependencies changed. As part of our future work, we plan to use more sophisticated heuristics. In RQ3, we only investigated calling relationships. Analogously, in future work, we plan to investigate other kinds of structural relationships. Moreover, even though we selected projects of different sizes and domains, the small number of subjects limits the generalizability of our results. Furthermore, we have only investigated a single module of a single branch of each project. Finally, given the research method we used, we cannot infer causal relationships between dependencies and co-changes.

4.4 Summary

Software development is an inherently complex socio-technical activity. Since the notorious software crisis acknowledged in the late 60's (Naur and Randell, 1969), practitioners and researchers have sought better ways to develop software systems. During this quest, some fundamental Software Engineering principles have emerged. One of these principles is known as "low coupling" and says that both the number and the strength of interconnections among modules should be minimized in order to prevent change propagation.

In this study, we set out to empirically investigate the link between dependencies and change propagation. With the support of a fairly sophisticated toolset, we analyzed 4 open source systems written in Java. Our results indicated that, in general, it is more likely that two artifacts will not co-change just because one depends on the other. However, the rate with which an artifact co-changes

with another is higher when the former structurally depends on the latter. This rate becomes higher if we track down dependencies to the low-level entities that are changed in commits. We also found several cases where software changes could not be justified using structural dependencies, meaning that co-changes are possibly induced by other subtler kinds of relationships

Besides addressing some of the limitations given in the prior section, as future work we plan and factorize the analysis done in RQ2 according to the different kinds of structural dependencies, including method calls, imports, and others. This would allow us to know the role of each kind of dependency separately.

Chapter 5

Dependencies and Software Changes: A Large-Scale Empirical Study

With the lessons learned from the warm-up study (Oliva and Gerosa, 2015), we designed a new study that investigates similar aspects, but at a much larger scale. More specifically, using a fairly sophisticated toolset, we mined structural dependencies and co-changes from 45 randomly chosen Java projects belonging to the Apache Software Foundation. This work extends the previous one by: (i) capturing dependencies more precisely using a branch-aware method, (ii) evaluating the statistical relationship between dependencies and co-changes using the chi-squared, and (iii) analyzing the importance of kind of dependency when trying to explain co-changes. In the following, we list our research questions and summarize the main results we obtained.

(RQ1) Are structurally dependent files more likely to co-change than independent ones?

Yes, dependencies increase the likelihood that two artifacts will co-change. When a class A depends on another class B and B changes, the likelihood that A will change together with B is 32% in average, being around 20% higher in average than the likelihood found in the case where A does not depend on B. Hence, the "low coupling" principle holds.

(RQ2) How accurately can co-changes be predicted using structural dependencies?

We recorded dependencies between pairs of files and built a classifier for each system using Random Forests. Classifiers had a poor accuracy in general, with Area Under the Curve (AUC) ranging from 0.52 to 0.76, thus implying that it is not possible to accurately predict co-changes solely using dependencies.

(RQ3) What kinds of structural dependencies best explain the occurrence of co-changes?

Analyzing the mean decrease in accuracy for each variable of the Random Forests model, we discovered that the length of indirect dependency, number of type imports, number of method calls, and number of references are the best predictors for co-changes. To the best of our knowledge, no change propagation prediction study has taken the length of indirect dependencies into account (either as a heuristic or a model variable).

Study organization. The rest of this study is organized as follows. In Section 5.1, we describe the study design, which includes our goals, the procedures for subject systems selection and data

collection, and the tools we used. In Section 5.2, we present the detailed results for the research questions. In Section 5.3, we discuss the results of the previous section. In Section 5.4, we discuss the threats to the validity of this study. Finally, in Section 5.5, we provide a summary of our findings.

5.1 Study Design

In this section, we present our goals, the rationale for selecting the subject systems, our data extraction and analysis approaches, and the supporting tools we used.

5.1.1 Goals

The goal of this empirical study is to understand and quantify the interplay between structural dependencies and co-changes. We take into account all kinds of structural dependencies shown in Section 2.2.1. Our perspective is mainly that of software developers, who are daily challenged to keep coupling under control while developing and maintaining software systems. The target systems of our study are 45 Java projects from the Apache Software Foundation. More details about the procedures for the selection of these systems are given in the next subsection.

5.1.2 Selection of Subject Systems

The subject systems were selected as a random sample of Apache Software Foundation (ASF) projects written in Java and stored in SVN. In the following, we show the rationale for these criteria.

Programming language. The detection of structural dependencies is done via static analysis, and thus depends on the programming language in which the target is written. We decided to focus our analysis on Java projects. According to the GitHub website (La, 2015), Java is second in the yearly rank of programming language popularity since 2014. More details about programming language use in GitHub can be found at the GitHut website (Zapponi, 2014). The large public directory of free and open source software OpenHub shows that Java ranks third in terms of the number of monthly commits in late 2015 when compared to C++, C#, JavaScript, Perl, PHP, Python, and Ruby (BlackDuck, 2016a). Hence, this study deals with subject systems written in a programming language that is often used in open-source development, making it relevant to a broad audience.

Version control system. Brindescu et al. (2014) conducted a large-scale empirical study (358k commits, 132 repositories, 5890 developers) and showed that commits made in distributed repositories were 32% smaller (fewer files) than in centralized repositories, and that developers split commits more often in distributed repositories. To minimize confounding factors, we decided to focus on a single distribution paradigm.

We decided to analyze projects from the ASF, which is a non-profit organization that has developed nearly a hundred distinguishing software projects that cover a wide range of technologies and address several problems from diverse contexts. In particular, the popularity of ASF projects and the heterogeneity of their projects made it a suitable choice for our study. In addition, ASF currently owns a single Subversion repository that hosts all Apache projects.

Subject systems selection. According to the Apache Projects Directory, most of their projects (215, 61.3%) are written in Java. In particular, the website stores high-level development metadata for each project, including the URL of the project's version control system. Relying on information

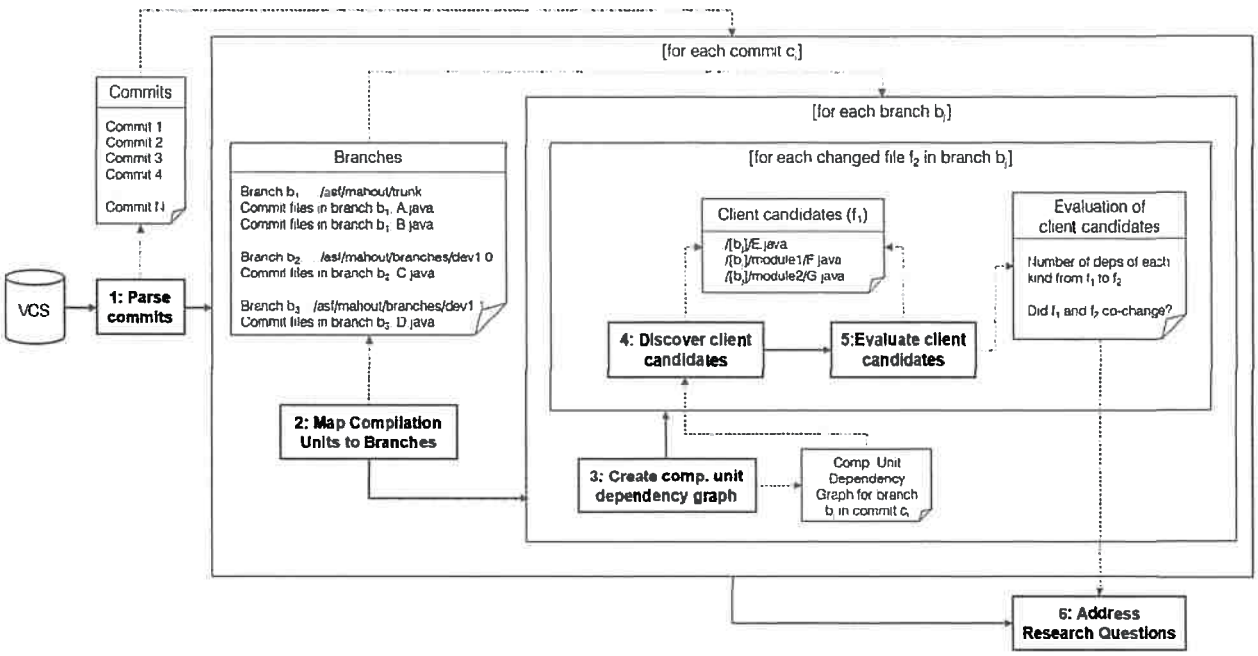

Figure 5.1: *Overview of study design*

from this webpage, we selected all projects written in Java and that had an associated Subversion URL (we did not validate these data for missing or incorrect information). We discovered that 165 out of the 215 Java projects had an associated Subversion URL. We regarded this set of projects as the population of this study. From this population, we obtained a random sample of 45 projects.

5.1.3 Approach

We mined the version control system of each subject system and, for each commit, check whether pairs of files (i) co-changed and/or (ii) a structural dependency existed from one file to the other. Our hypothesis is that a file `A` should co-change more frequently with `B` when it depends on `B` (as compared to when `A` does not depend on `B`). An overview of the approach is depicted in Figure 5.1. In the following, we describe each of the main steps we followed.

Parse Commits

All Apache Software Foundation projects in Subversion are hosted under the same repository. As of November 9th, 2015, this repository hosts more than 1.71 million commits. Working with large remote repositories poses a series of challenges. To cope with them, we have maintained a local copy that we synchronize in an on-demand fashion. By the time we wrote this study, our mirror pointed to commit 1,615,288.

We used XFlow (Santana et al., 2011) to collect, parse, and store commit metadata of each subject project. The input consisted of two parameters: the version control system URL and the commit range (*first commit*, *last commit*) to be analyzed. We also used the tool to perform two filters on-the-fly. First, we only processed commits that included at least one Java file. Second, we discarded commits having more than 100 Java files, since these usually refer to repository operations (e.g., branch merging) and crosscutting changes (e.g., license change) - both of which have little to do with structural dependencies. In addition, these commits would require non-justifiable high computational power to be processed (Table 5.3).

Map Compilation Units to Branches

After we parsed the project's commits, we processed one commit at a time (note the outer large rectangle in Figure 5.1). For each commit, we discovered the branches that the commit modifies. Before we dig into how we discovered these branches, let us first justify the effort. For each added or modified Java file, we need to know which branch it belongs to in order to perform a fair analysis of structural dependencies and co-changes. The main reason is that development branches are often created in the repository to store and evolve an independent version of the system. One cannot expect a branch to have compilation units that depend on other compilation from other branches (e.g., a file `A` from the trunk should not depend on a file `B` from a sandbox branch). Therefore, our whole approach is branch-aware, in the sense that it does not evaluate pairs of files from different branches for analyzing structural dependencies and co-changes. In fact, our approach is also module-dependency-aware, as we shall see in step 4.

We discovered the branches that a certain commit modifies by using a mix of heuristics and manual validation. Finally, once we detected all branches that existed in the project's repository at a particular commit, we mapped the Java files in the commit's change-set to those branches.

Create Compilation Unit Dependency Graph

In this step, we processed the branches discovered in the previous step, one at a time. For each branch, we checked-out its code from the repository and detected the structural dependencies using JDX (see Section 5.1.4 for a description of this tool). In other words, the dependency extraction takes into account only the files in the specific branch being processed. Once dependencies were captured, we built a Compilation Unit Meta-Dependency Graph.

This is a directed graph where nodes are compilation units and edges are meta-dependencies. A meta-dependency indicates the existence of one or more underlying dependencies between two compilation units. For instance, a meta-dependency from `A.java` to `B.java` could indicate that a certain method `A.foo()` calls `B.bar()`, that `A` uses `B` as a method parameter in `A.do(B)`, and that `A` imports `B`. We opted to build this graph because it is smaller (in terms of both the number of nodes and edges) and thus easier to work with. Furthermore, as we stated, we can recover the specific dependencies from each meta-dependency. We relied on this graph to discover whether a certain compilation unit `A` depends on another unit `B`.

Discover Client Candidates

For each modified file, we needed to discover which other files depended on it. As we noted in step 2, our approach is branch-aware, so only files from the same branch are taken into account. In addition to that, our approach is also module-dependency-aware. The reason is that projects often indicate allowed module dependencies in their build files. As a consequence, only files from a subset of modules (client modules) are allowed to depend on a certain file from a specific module (provider module). Therefore, for each modified file fj, we needed to discover which other files fi were allowed to depend on it (client candidates).

Parsing the build files to extract these pre-established module dependencies would require developing a parser for each build technology. Given these problems, we conceived and applied an alternative approach, which is agnostic to the build technology the projects use. We first discovered

all the modules of the branch being processed by inferring them from the path of the compilation units in that branch. Afterwards, we lifted the compilation unit meta-dependency graph built in step 3 to derive the module dependencies. For instance, if a compilation unit `A` from module examples depends on a compilation unit from module utils, then we say that examples depends on utils. After that, we created a Module Dependency graph, in which nodes represent modules and edges represent dependencies among these modules.

The rationale is that this graph should correspond to the dependencies established in the build file. If there is a dependency from module `A` to module `B` in the build file and files from module `A` indeed call files from module B, then this module dependency will appear in our graph. If a module dependency is listed in the build file but no actual dependencies between files from these modules exist, then there will be no dependencies between the modules, and thus no dependencies between these modules will show up in our graph. Finally, if a dependency from module `A` to `B` is not listed in the build file, developers will not be able to make calls from A's files to B's files, yielding no dependencies between the two modules. In this case, our graph will have no dependencies between these modules.

Therefore, given a certain modified file fj, the client candidates were all those lying in fj's module and in other modules that depended on fj's module.

Evaluate Client Candidates

In this step, each client candidate f_1 in the checked-out code was evaluated against a file f_2 in the commit's change-set. This evaluation consisted of determining which dependencies existed (if any) from f_1 to f_2 and whether the files co-changed. We highlight that, at this point, we have ensured that f_1 and f_2 belonged to the same branch and that the module from f_1 was allowed to call the module from f_2.

Dependencies from f_1 to f_2 were straightforwardly derived from the compilation unit meta-dependency graph. We counted how many dependencies of each kind (Section 2.2.1) existed from f_1 to f_2. We also checked whether f_1 indirectly depended on f_2. More specifically, we counted how many chained dependencies it takes to connect f_1 to f_2. If no such path exists, we registered the value zero.

Co-change was determined by checking whether both files were included in the commit's change-set. The results of this evaluation were incrementally registered in a table and then later on used to answer both research questions. From now on, we call this table the dependencies dataset.

Table 5.1 shows an excerpt of a hypothetical dependencies dataset. It focuses on the results of processing commit #10 from a certain project. The commit changed two files, namely: A.java and B.java. The project has only 4 classes at this point (commit 10), namely: A.java, B.java, C.java, and D.java. To simplify the example, we assume that all four classes lie in the same branch (e.g., trunk) and in the same module. The results of processing commit #10 are indicated in rows 151-155. Row 151 indicates that, in the code snapshot corresponding to commit 10, A.java calls methods of B.java five times, and that the former has a method that returns an instance of latter type. As A.java and B.java are included in the change-set of commit 10, we mark that they co-changed. In the second row, we notice that even though there were no direct dependencies from C.java to A.java, there was an indirect dependency of length 3, i.e., it took 3 direct dependencies (hops) to reach `A` from C. The exact path is not registered. The third row shows an example where there is no dependency from

Table 5.1: *Excerpt of a hypothetical dependencies dataset*

Row	Commit	f_1 (checked out file)	f_2 (changed file)	#Access Deps.	#Method Call Deps.	#Param Deps.	#Ref Deps.	#Return Deps.	#Throw Deps.	#Class Inh. Deps.	#Interf. Inh. Deps	#Type Import Deps.	#Method Import Deps.	Length Indirect Dep.	Co-Change
150	09	...	...	...	...	...	...	...	...	...	...	...	...	...	...
151	10	B.java	A.java	0	5	0	0	1	0	0	0	0	0	1	YES
152	10	C.java	A.java	0	0	0	0	0	0	0	0	0	0	3	NO
153	10	D.java	A.java	0	0	0	0	0	0	0	0	0	0	0	NO
154	10	C.java	B.java	0	0	0	0	0	0	1	0	0	0	1	NO
155	10	D.java	B.java	0	0	0	0	0	0	0	0	0	0	0	NO
156	11	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Table 5.2: *Relationship Status*

File pair $\langle f_1, f_2 \rangle$		**Does f_1 depend on f_2?**	
		Yes (there is at least one dep. from f_1 to f_2	No (there are no deps. from f_1 to f_2)
Does f_1 co-change with f_2?	Yes (f_2 changes and f_1 changes as well)	CD	$C\overline{D}$
	No (f_2 changes and f_1 *does not* change)	$\overline{C}D$	$\overline{CD}$

f_1 to f2, even in the indirect case. The fourth row indicates that a type from C.java extends a type from B.java, but that the two files did not co-change in commit 10. Finally, the last row is another example where f_1 does not depend on f_2 and the two files do not co-change. This is by far the most common scenario.

Row 150 indicates the last $\langle f_1, f_2 \rangle$ pair evaluated as part of the analysis of commit 9. Analogously, row 156 indicates the first pair evaluation as part of the analysis of commit 11. This is meant to exemplify that this table is constructed incrementally and that it registers the results of every pair evaluation of every commit in the scope of analysis. Finally, we highlight that the same pair $\langle f_1, f_2 \rangle$ might appear in several rows of this table, as the same file f_2 might be changed in several commits and f_1 might exist in the code snapshot corresponding to all these commits.

Address Research Questions

Addressing RQ1. In this research question, we aim to discover whether dependent files are more likely to co-change than independent ones. In statistical terms, we want to discover whether there is an association between a variable denoting the presence (or absence) of structural dependencies and a variable representing the occurrence (or not) of co-changes. To this end, we derived n contingency tables summarizing the evaluation results of the n distinct $\langle f_1, f_2 \rangle$ pairs included in Table 5.1. A contingency table is a data structure that shows the distribution of one variable in rows and another in columns. The contingency table we elaborated for a given $\langle f_1, f_2 \rangle$ pair takes the form depicted in Table 5.6. The four cells register how many times each of the four possible combinations occurred: co-change and (at least one) dependency (CD), co-change and no dependency ($C\overline{D}$) no co-change and at least one dependency ($\overline{C}D$) , and no co-change and no-dependency ($\overline{CD}$).

To answer the research question, we created a new single contingency table C that covered the whole evolution of the system by aggregating the data from all the n contingency tables. Each cell $C[i,j]$, with i,j in $\{1,2\}$, was defined as follows: $C[i,j] = \sum_{p=1}^{n} C_p[i,j]$, where C_p is the contingency

table for a pair $p = \langle f_1, f_2 \rangle$ and n is the total number of contingency tables. In other words, each cell of C is the sum of the corresponding cells from all n contingency tables we created. Once we built C, we ran Pearson's Chi-Squared test of independence with $\alpha = 0.05$. This test assesses whether unpaired observations on two variables, expressed in a contingency table, are independent of each other. In our case, the two variables were presence of structural dependencies and occurrence of co-change.

Addressing RQ2 and RQ3. In this research question, we aim to discover the kinds of structural dependencies that best help to explain the occurrence of co-changes. To this end, we built a classification model. In the domain of statistical learning, supervised learning involves building a statistical model for predicting an output based on one or more inputs (James et al., 2013). Classification is an instance of supervised statistical learning, in which the goal is to predict a qualitative response for a given observation. A classification model tackles a certain classification problem. In our study, we built a classification model to predict whether files will co-change based on structural dependencies information. Ultimately, we investigate this model to understand which predictors (kinds of structural dependencies) best the occurrence of co-changes. We build the input to our classification model out of the dependencies dataset (Table 5.1) as follows: each dependency column was taken as a feature/explanatory variable (e.g., the number of call dependencies) and the occurrence (or absence) of co-changes was taken as the class under observation.

We prepared this input dataset using a state-of-the-art approach (McIntosh et al., 2015; Tantithamthavorn et a 2015), which includes (i) removing highly correlated features and (ii) removing redundant variables. The goal of the first step is to reduce multicollinearity among the features. We chose to use Spearman's rank correlation test (ρ) instead of other types of correlation (e.g., Pearson's) because it is resilient to data that is not normally distributed. We employed a variable clustering analysis known as *varclus*, which produces a hierarchical overview of the correlation among all explanatory variables. As in the work of McIntosh and colleagues McIntosh et al. (2015), for subhierarchies of explanatory variables with correlation $|\rho| > 0.7$, we selected only one variable from the subhierarchy for inclusion in the model. We used the `varclus` implementation provided by the `Hmisc` R package (Harrell Jr., 2016b). This package was written by Frank Harrell Jr., an expert in regression modeling techniques (Harrell Jr., 2015).

Even though correlation analysis reduces multicollinearity, it may not detect all of the redundant variables, i.e., variables that do not have a unique signal from the other explanatory variables. Redundant variables interfere with each other in an explanatory model, "distorting the modeled relationship between the explanatory and dependent variables" (McIntosh et al., 2015). Hence, we removed redundant variables from our dataset. The approach we used consisted of fitting preliminary models that explain each feature using the other available features. We then used the R^2 value of the preliminary models to measure and decide how well each feature was explained by the others. As in literature (McIntosh et al., 2015; Tantithamthavorn et al., 2015), we used the implementation provided by the `redun` function from the `Hmisc` R package (Harrell Jr., 2016a) . This function builds preliminary models for each feature in each bootstrap iteration. The feature that is most well-explained by the others is iteratively dropped until either: (1) no preliminary model achieves an R^2 above a cutoff threshold, or (2) removing a metric would make a previously dropped metric no longer explainable, i.e., its preliminary model will no longer achieve an R^2 exceeding the threshold (McIntosh et al., 2015; Tantithamthavorn et al., 2015). For this study, we used the default threshold

of 0.9.

After the dataset was prepared, we built the classification model using Random Forests (Breiman, 2001). A Random Forest is an accurate classification technique that has been used in several literature studies (Fukushima et al., 2014; Gousios et al., 2014; Lessmann et al., 2008; Tantithamthavorn et al., 2015). As described by James et al. (2013), the technique involves building a collection of decision trees, where each tree gets a "vote" in classifying. Random forests have two main components of randomness. For each tree, the algorithm selects a random subsample of the dataset to build it (a.k.a., bootstrapped training samples). In addition, at each split during decision tree building, the algorithm chooses a fresh random sample of predictors (features) and uses only one from this sample. This ensures a low correlation degree between the predictions made by the trees. A final decision is made by aggregating the votes from all trees and comparing the fraction of votes in each class to a certain threshold. We followed the common practice, which is to take the wining class as the one with the majority of the votes. We used the implementation of Random Forests provided by the `randomForest` R package (Breiman and Cutler, 2015). After the classification model was built, we answered RQ2 by evaluating the classifier's performance using Area Under the Curve (AUC) and the out-of-bag error (OOB).

In order to answer RQ3, we analyzed variables' importance in the Random Forests model. Analogously to Tantithamthavorn et al. (2015), we relied on the technique introduced by Breiman (2001), which is implemented in the `randomForest` R package (Breiman and Cutler, 2015) via the `importance` function. As explained in the package's manual, "for each tree, the prediction error on the out-of-bag portion of the data is recorded (error rate). Then the same is done after permuting each predictor variable. The difference between the two are then averaged over all trees, and normalized by the standard deviation of the differences." The intuition is that, if the variable is important, disturbing it will cause a great impact in the classifier's accuracy.

5.1.4 Supporting Tools

The supporting tools we used are exactly the same as those employed in the warm-up study (Section 4.1.6). However, we extended the Structural Dependencies Analyzer, so that it is now more abstract and acts like a framework that can be used for several studies relying on a historical analysis of structural dependencies. In addition, for this study, we used additional packages in R, namely: `varclus` (variable correlation), `redun` (variable redundancy), and `randomForest` (implementation of Random Forests).

5.2 Study Results

After applying the procedures described in Section 5.1.2, we obtained a random sample of 45 projects from the Apache Software Foundation. Table 5.3 lists the projects and some of their characteristics, including: the number and date of first and last commits, project age in years, number of commits, and number of developers who committed at least once (commiters). Subject systems differ substantially in terms of the domains they tackle. Data from Table 5.3 also show that they differ in terms of age (1.5 to 9.4 years), number of commits (292 to 18,776), and number of commiters (1 to 37). Therefore, the random selection of projects led to a heterogeneous sample that resembles the inherent heterogeneity of the whole population. Data from OpenHub also supports this claim

(BlackDuck, 2016b).

In our study, we discard commits with 100 files or more, as then often point to crosscutting operations (repository merges, etc.) and do not reflect true change propagations. Table 5.4 presents the number of commits we obtained after applying this filter.

Figure 5.2 shows that the number of compilation units increased over time. This was expected, as systems tend to grow over time (Lehman et al., 1997). Interestingly, the average number of suppliers also increased over time, as shown in Figure 5.3.

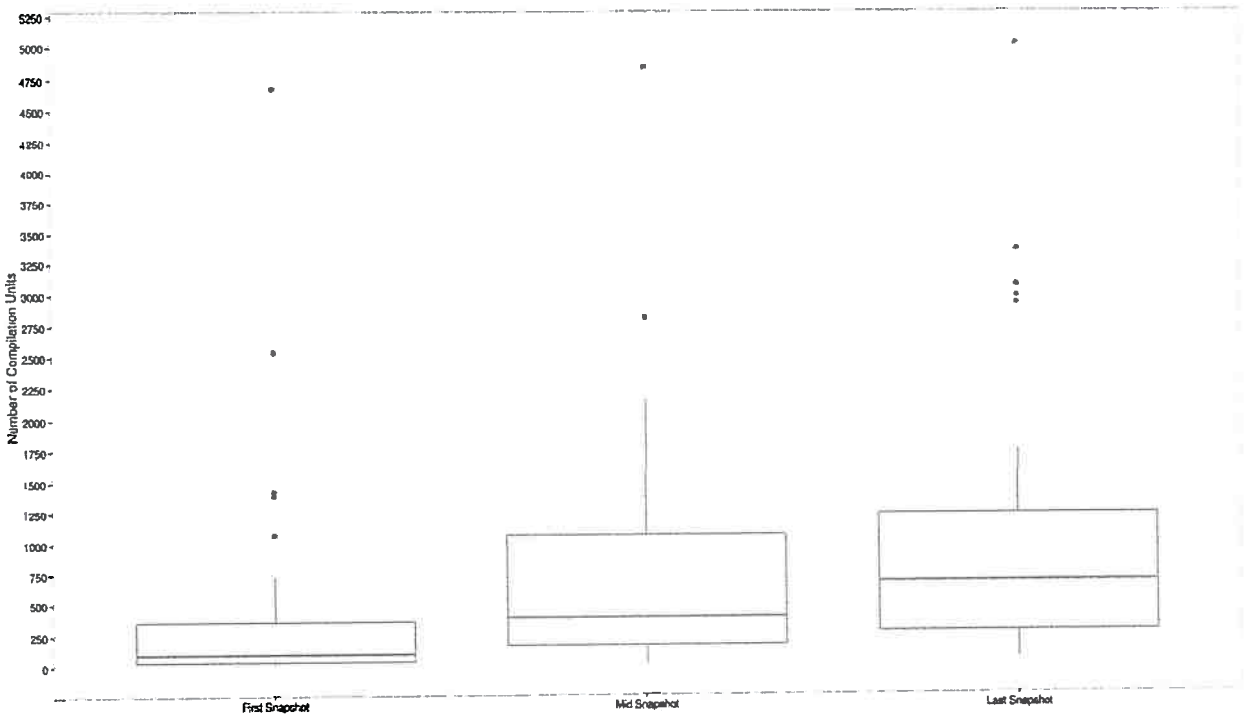

Figure 5.2: *Evolution of the number of compilation units*

In Table 5.5, we show the number of branches in each system and compare the number of *trunk* branches and *non-trunk* branches. As we can easily notice, some projects have a very high number of branches (118 for ManifoldCF). We also notice that some project move the location of their trunk folder frequently. This all reinforces the need to perform an analysis that covers all branches, since analyzing only a single trunk branch could lead to misleading results.

5.2.1 RQ1: Are Structurally Dependent Files More Likely To Co-Change Than Independent Ones?

As detailed in Section 5.1.3, we mined the version control system of each subject system and, for each commit, determined whether pairs of files $\langle f_1, f_2 \rangle$ co-changed and whether at least one structural dependency existed from f_1 to f_2. We also recall that f_1 is a file from the checked out code and f_2 is a file from the commit's change-set. Grounded on Software Engineering literature (Fowler, 2001; Larman, 2004; Martin and Martin, 2006), our hypothesis is that a file `A` should co-change more frequently with `B` when it depends on it (as compared to when `A` does not depend on `B`).

We derived a single contingency table C for the whole system from all k contingency tables. Afterwards, we ran the chi-squared (χ^2) independence test to discover whether the "presence of

Table 5.3: *Description of Subject Systems*

Project	SVN URL	First Commit	Date of First Commit	Last Commit	Date of Last Commit	Age (years)	Commits	Devs
Etch	[root]/etch	712730	29/07/2008	1604167	20/06/2014	5.9	714	8
Commons Modeler	[root]/commons/proper/modeler	139064	27/01/2005	1534259	21/10/2013	8.7	333	15
ECS (in the Attic)	[root]/jakarta/ecs	169673	06/05/2005	1030542	03/11/2010	5.5	292	14
SSHD	[root]/mina/sshd	723367	04/12/2008	1434658	17/01/2013	4.1	320	4
Scout	[root]/juddi/scout	57050	09/11/2004	1413669	26/11/2012	8.0	487	8
Oltu - Parent	[root]/oltu	949823	31/05/2010	1605764	26/06/2014	4.1	695	7
Excalibur (in the Attic)	[root]/excalibur	21165	13/06/2004	1053268	28/12/2010	6.5	1284	19
Commons DbUtils	[root]/commons/proper/dbutils	141653	27/01/2005	1593218	08/05/2014	9.3	640	14
DirectMemory	[root]/directmemory	1179158	05/10/2011	1604957	23/06/2014	2.7	819	8
Commons Proxy	[root]/commons/proper/proxy	234282	21/08/2005	1585561	07/04/2014	8.6	643	7
Axiom	[root]/webservices/axiom	1031208	04/11/2010	1607207	01/07/2014	3.7	514	1
Hivemind (in the Attic)	[root]/hivemind	188858	07/06/2005	1030543	03/11/2010	5.4	932	8
Wink	[root]/wink	779148	27/05/2009	1605853	26/06/2014	5.1	1319	11
Commons CLI	[root]/commons/proper/cli	129765	27/01/2005	1570439	21/02/2014	9.1	1001	20
Vysper	[root]/mina/vysper	537105	11/05/2007	1423331	18/12/2012	5.6	963	8
ACE	[root]/ace	772975	08/05/2009	1598208	29/05/2014	5.1	2072	8
Woden	[root]/webservices/woden	160916	11/04/2005	1561765	27/01/2014	8.8	1448	12
Velocity Tools	[root]/velocity/tools	71525	15/11/2004	1589329	22/04/2014	9.4	1590	11
Hama	[root]/hama	659418	23/05/2008	1607244	02/07/2014	6.1	1629	14
TomEE	[root]/tomee/tomee	1434793	17/01/2013	1607123	01/07/2014	1.5	1669	7
Commons IO	[root]/commons/proper/io	140283	27/01/2005	1603493	18/06/2014	9.4	1783	23
Commons Compress	[root]/commons/proper/compress	144651	27/01/2005	1604313	21/06/2014	9.4	1603	14
Commons Net	[root]/commons/proper/net	139270	27/01/2005	1601150	07/06/2014	9.4	1901	15
cTAKES	[root]/ctakes	1385218	16/09/2012	1606243	27/06/2014	1.8	2993	21
HttpComponents Core	[root]/httpcomponents/httpcore	392762	09/04/2006	1606475	29/06/2014	8.2	2155	8
Chemistry	[root]/chemistry	770097	30/04/2009	1606578	29/06/2014	5.2	3285	15
Torque	[root]/db/torque	227527	04/08/2005	1586449	10/04/2014	8.7	4486	21
Pig	[root]/pig	1003920	02/10/2010	1606892	30/06/2014	3.7	2710	21
PDFBox	[root]/pdfbox	620303	10/02/2008	1607237	01/07/2014	6.4	2600	16
Beehive (in the Attic)	[root]/beehive	22922	15/07/2004	928328	28/03/2010	5.7	3506	15
Stanbol	[root]/stanbol	1035261	15/11/2010	1605683	26/06/2014	3.6	4584	20
Commons Configuration	[root]/commons/proper/configuration	141754	27/01/2005	1606592	29/06/2014	9.4	2817	14
Aries	[root]/aries	817817	22/09/2009	1607004	01/07/2014	4.8	5531	32
OODT	[root]/oodt	902889	25/01/2010	1607211	01/07/2014	4.4	4284	19
Mahout	[root]/mahout	612001	15/01/2008	1605860	26/06/2014	6.4	3758	24
ManifoldCF	[root]/manifoldcf	898094	11/01/2010	1607170	01/07/2014	4.5	5026	8
Shindig	[root]/shindig	601882	06/12/2007	1605175	24/06/2014	6.6	5344	34
Continuum	[root]/continuum	160813	10/04/2005	1605752	26/06/2014	9.2	6608	21
Hive	[root]/hive	1005672	07/10/2010	1607220	01/07/2014	3.7	3980	35
Synapse	[root]/synapse	234477	22/08/2005	1561484	26/01/2014	8.4	4689	29
JAMES	[root]/james/server	107010	30/11/2004	1592312	04/05/2014	9.4	7582	32
Lenya (in the Attic)	[root]/lenya	37790	03/09/2004	1549014	08/12/2013	9.3	13440	37
UIMA	[root]/uima	469325	30/10/2006	1607171	01/07/2014	7.7	12845	18
Xerces for Java	[root]/xerces/java	315084	12/10/2005	1594899	15/05/2014	8.6	7697	29
Qpid	[root]/qpid	443185	13/09/2006	1607176	01/07/2014	7.8	18776	34

Table 5.4: *Subject systems - Commit numbers*

Project	Commits	Commits with Java Files	Commits with less than 100 Java Files
Etch	714	144 (20.2%)	140 (97.2%)
Commons Modeler	333	157 (47.1%)	156 (99.4%)
ECS (in the Attic)	292	177 (60.6%)	174 (98.3%)
SSHD	320	211 (65.9%)	207 (98.1%)
Scout	487	214 (43.9%)	213 (99.5%)
Oltu - Parent	695	246 (35.4%)	244 (99.2%)
Excalibur (in the Attic)	1284	271 (21.1%)	266 (98.2%)
Commons DbUtils	640	293 (45.8%)	291 (99.3%)
DirectMemory	819	322 (39.3%)	321 (99.7%)
Commons Proxy	643	395 (61.4%)	393 (99.5%)
Axiom	514	397 (77.2%)	394 (99.2%)
Hivemind (in the Attic)	932	554 (59.4%)	541 (97.7%)
Wink	1319	562 (42.6%)	550 (97.9%)
Commons CLI	1001	564 (56.3%)	560 (99.3%)
Vysper	963	771 (80.1%)	767 (99.5%)
ACE	2072	792 (38.2%)	786 (99.2%)
Woden	1448	836 (57.7%)	831 (99.4%)
Velocity Tools	1590	888 (55.8%)	888 (100.0%)
Hama	1629	1009 (61.9%)	1003 (99.4%)
TomEE	1669	1179 (70.6%)	1168 (99.1%)
Commons IO	1783	1187 (66.6%)	1183 (99.7%)
Commons Compress	1603	1192 (74.4%)	1191 (99.9%)
Commons Net	1901	1215 (63.9%)	1201 (98.8%)
cTAKES	2993	1316 (44.0%)	1300 (98.8%)
HttpComponents Core	2155	1549 (71.9%)	1538 (99.3%)
Chemistry	3285	1759 (53.5%)	1748 (99.4%)
Torque	4486	1829 (40.8%)	1822 (99.6%)
Pig	2710	1885 (69.6%)	1876 (99.5%)
PDFBox	2600	1888 (72.6%)	1879 (99.5%)
Beehive (in the Attic)	3506	1932 (55.1%)	1912 (99.0%)
Stanbol	4584	2064 (45.0%)	2048 (99.2%)
Commons Configuration	2817	2108 (74.8%)	2101 (99.7%)
Aries	5531	2157 (39.0%)	2143 (99.4%)
OODT	4284	2164 (50.5%)	2148 (99.3%)
Mahout	3758	2270 (60.4%)	2226 (98.1%)
ManifoldCF	5026	2365 (47.1%)	2345 (99.2%)
Shindig	5344	2479 (46.4%)	2460 (99.2%)
Continuum	6608	2976 (45.0%)	2947 (99.0%)
Hive	3980	2997 (75.3%)	2952 (98.5%)
Synapse	4689	3011 (64.2%)	3001 (99.7%)
JAMES	7582	4463 (58.9%)	4421 (99.1%)
Lenya (in the Attic)	13440	4633 (34.5%)	4608 (99.5%)
UIMA	12845	5498 (42.8%)	5443 (99.0%)
Xerces for Java	7697	6110 (79.4%)	6100 (99.8%)
Qpid	18776	6257 (33.3%)	6152 (98.3%)

Table 5.5: *Number of branches per project*

Project	#Branches	#Trunk Branches	Percentage of Commits in Trunk Branches	#Other Branches	Percentage of Commits in Other Branches
ACE	4	1	96.4%	3	3.6%
Aries	16	1	90.2%	15	9.8%
Axiom	14	1	62.4%	13	37.6%
Behive	10	2	98.5%	8	1.5%
Chemistry	14	7	97.2%	7	2.8%
Commons CLI	8	1	50.8%	7	49.2%
Commons Compress	5	1	90.5%	4	9.5%
Commons Configuration	5	1	61.2%	4	38.8%
Commons DbUtils	2	1	74.0%	1	26.0%
Commons IO	3	1	99.9%	2	0.1%
Commons Modeler	1	1	100.0%	0	0.0%
Commons Net	7	1	78.3%	6	21.7%
Commons Proxy	3	1	74.0%	2	26.0%
Continuum	28	1	65.0%	27	35.0%
cTakes	17	1	87.6%	16	12.4%
Direct Memory	2	2	100.0%	0	0.0%
ECS	1	1	100.0%	0	0.0%
Etch	12	1	67.1%	11	32.9%
Excalibur	3	1	94.6%	2	5.4%
Hama	8	1	95.2%	7	4.8%
Hive	14	1	66.3%	13	33.7%
Hivemind	6	2	73.0%	4	27.0%
HttpComponents Core	6	1	96.4%	5	3.6%
James	20	2	71.1%	18	28.9%
Lenya	17	2	84.6%	15	15.4%
Mahout	2	1	100.0%	1	0.0%
ManifoldCF	118	5	49.5%	113	50.5%
Oltu Parent	3	1	99.8%	2	0.2%
OODT	51	19	75.5%	32	24.5%
PDFBox	7	3	80.8%	4	19.2%
Pig	8	1	55.5%	7	44.5%
Qpid	64	3	48.0%	61	52.0%
Scout	5	1	70.2%	4	29.8%
Shindig	7	2	96.8%	5	3.2%
SSHD	1	1	100.0%	0	0.0%
Stanbol	18	1	89.7%	17	10.3%
Synapse	13	1	94.1%	12	5.9%
TomEE	5	1	99.3%	4	0.7%
Torque	12	7	95.8%	5	4.2%
UIMA	24	12	95.5%	12	4.5%
Velocity Tools	5	1	47.7%	4	52.3%
Vysper	8	1	63.9%	7	36.1%
Wink	10	3	97.3%	7	2.7%
Woden	12	1	41.8%	11	58.2%

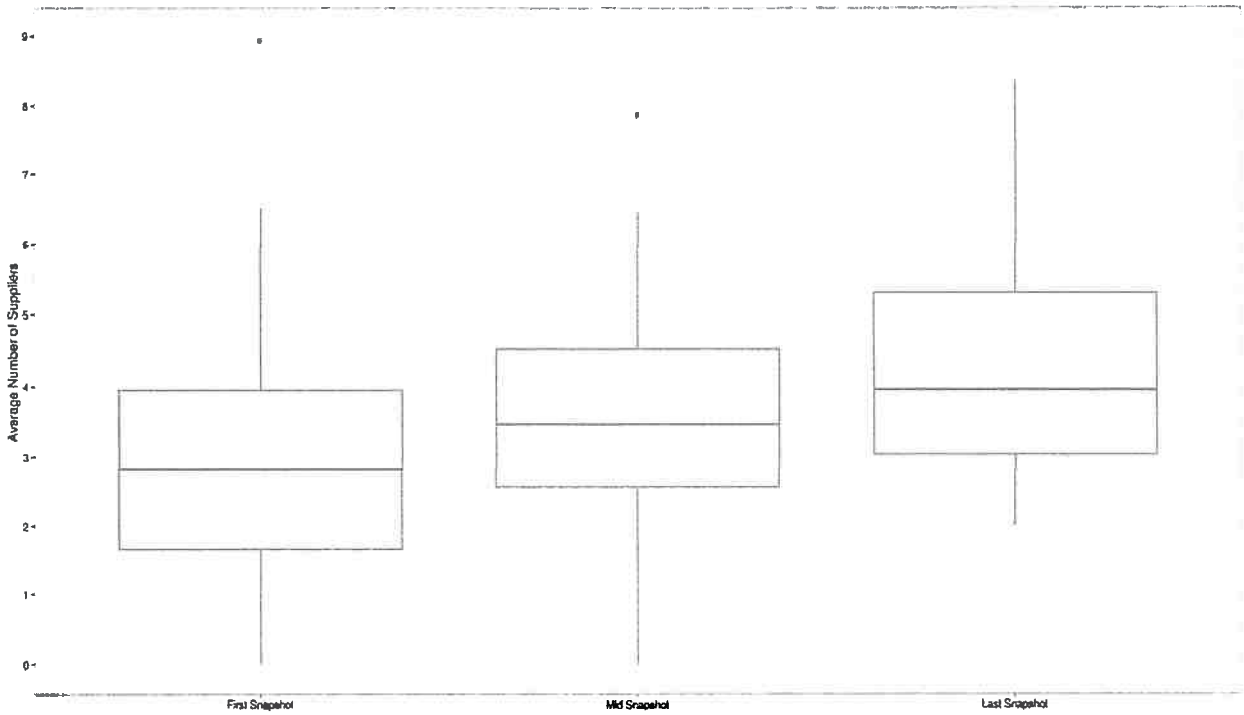

Figure 5.3: *Evolution of the average number of suppliers per compilation unit*

dependencies" and the "occurrence of co-changes" are associated. To support interpreting the results, we also calculated two metrics, which are defined as follows:

$CoChangeRatioWithDep(f_1, f_2) = \frac{CD}{CD+\overline{CD}} = R_1$ the number of commits in which f_1 depended on f_2 and f_1 co-changed with f_2 over the number of commits in which f_2 changed and f_1 depended on f_2.

$CoChangeRatioWithoutDep(f_1, f_2) = \frac{\overline{CD}}{\overline{CD}+\overline{\overline{CD}}} = R_2$ the number of commits in which f_1 did not depend on f_2 and f_1 co-changed with f_2 over the number of commits in which f_2 changed and f_1 did not depend on f_2.

The results we obtained are depicted in Table V. The Chi-Squared test revealed an association between the two variables in almost all projects. Furthermore, R_1 was almost always higher than R_2. In particular, the median of the former was 31.7% and the median of the latter was 9.7%. To determine if this difference between the medians was statistically significant, we performed a paired two-sample Wilcoxon test (a.k.a., Mann-Whitney test) with the alternative hypothesis that R_1 is greater than R_2 with $alpha = 0.05$. We could reject the null hypothesis, as the obtained p-value was 9.76×10^{-4}. Therefore, the co-change ratio is higher when there is dependency (as compared to when there is no dependency).

When a class A depends on another class B and B changes, the likelihood that A will change together with B is 32% in average, being around 20% higher in average than the likelihood found in the case where A does not depend on B. Hence, the "low coupling" principle holds.

Table 5.6: *Evaluation of Contingency Table*

Project	Contingency Table				R_1	R_2	χ^2 test
	CD	$C\overline{D}$	$\overline{C}D$	$\overline{CD}$			(*p-value*)
Hama	5304	37488	8822	368094	37.50%	9.24%	0
Ace	3666	39050	3172	133691	53.60%	22.60%	0
Aries	7068	64306	7603	319746	48.20%	16.70%	0
Hive	24265	236435	190464	16177392	11.30%	1.44%	0
Hivemind	5132	56147	7514	661109	40.60%	7.83%	0
HttpComponents Core	4967	58962	14670	936681	25.30%	5.92%	0
Axiom	3662	25814	20873	814778	14.90%	3.07%	0
James Server	15782	194967	24489	2782294	39.20%	6.55%	0
Beehive	13679	163548	30398	1809702	31%	8.29%	0
Lenya	12817	135716	45186	3243114	22.10%	4.02%	0
Chemistry	14904	137054	36324	1102851	29.10%	11.10%	0
Mahout	10354	173777	25473	4873507	28.90%	3.44%	0
ManifoldCF	6938	47329	25453	1239229	21.40%	3.68%	0
Oltu	541	3797	1074	12350	33.50%	23.50%	$< 7.11E-19$
OODT	13856	172852	12627	647832	52.30%	21.10%	0
PDFBox	7609	68356	62781	2610704	10.80%	2.55%	0
Pig	5590	55738	47355	4879911	10.60%	1.13%	0
Qpid	45117	446160	181597	13564897	19.90%	3.18%	0
Scout	713	9799	676	38392	51.30%	20.30%	$< 1.76E-170$
Shindig	8126	79080	21720	1563223	27.20%	4.82%	0
SSHD	1392	14231	2346	99740	37.20%	12.50%	0
Stanbol	14010	145279	16308	535767	46.20%	21.30%	0
Synapse	7364	154866	10325	3162107	41.60%	4.67%	0
TomEE	2199	33673	16354	2377924	11.90%	1.40%	0
Torque	5140	65737	10129	338063	33.70%	16.30%	0
UIMA	19068	302800	96177	5944918	16.50%	4.85%	0
Velocity	920	17496	2913	121378	24%	12.60%	$< 1.23E-95$
Vysper	3250	22835	10181	535299	24.20%	4.09%	0
Commons CLI	2013	17221	1972	31526	50.50%	35.30%	$< 1.36E-81$
Wink	1544	15800	4238	387661	26.70%	3.92%	0
Woden	4000	46463	6609	432847	37.70%	9.69%	0
Xerces	23014	260372	83272	10379910	21.70%	2.45%	0
Commons Compress	1588	19867	3221	158888	33%	11.10%	0
Commons Configuration	2345	16445	10508	416338	18.20%	3.80%	0
Commons DbUtils	395	4551	287	9106	57.90%	33.30%	$< 1.81E-39$
Commons IO	996	28846	2564	108916	28%	20.90%	$< 3.73E-24$
Commons Modeler	406	4814	548	5521	42.60%	46.60%	< 0.018772329
Commons Net	1722	41681	6322	399512	21.40%	9.45%	$< 3.20E-283$
Commons Proxy	1113	21095	920	27736	54.70%	43.20%	$< 1.04E-24$
Continuum	8397	107951	18086	1004388	31.70%	9.70%	0
cTAKES	2952	83267	3934	555241	42.90%	13%	0
DirectMemory	1068	7728	1298	45427	45.10%	14.50%	0
ECS	670	42473	9348	161831	6.69%	20.80%	$< 1.23E-258$
Etch	777	6951	825	25051	48.50%	21.70%	$< 4.52E-136$
Excalibur	1474	13377	2772	79763	34.70%	14.40%	$< 1.14E-284$

Table 5.7: *Summary of Classification Dataset*

Project	Rows	Rows with dependencies	No Co-change	Co-change
bval	190,155	24,073 (12.66%)	0.74	0.26
commons	180,402	9,291 (5.15%)	0.75	0.25
derby	2.7E+07	9,352,791 (35.17%)	0.98	0.02
mahout	6,617,325	211,782 (3.20%)	0.86	0.14
scout	47,946	5,793 (12.08%)	0.66	0.34
tentacles	1,628	161 (9.89%)	0.40	0.60
tomcat	4,950,500	551,840 (11.15%)	0.98	0.02
tomEE	2,378,300	286,386 (42.89%)	0.98	0.02
velocity	585,567	251,157 (42.89%)	0.91	0.09
whirr	48,575	12,836 (26.43%)	0.83	0.17

Table 5.8: *Results of Co-Change Prediction using Structural Dependencies*

Project	OOB Error	No Co-Change Class Error	Co-Change Class Error	AUC
bval	26.10%	0.46%	97.50%	0.54
commons	23.29%	1.61%	89.88%	0.65
derby	1.52%	0.01%	98.95%	0.53
mahout	13.98%	0.30%	96.85%	0.56
scout	29.92%	8.65%	71.74%	0.64
tentacles	22.98%	36.92%	13.54%	0.76
tomcat	2.41%	0.01%	99.26%	0.56
tomEE	2.33%	0.01%	99.83%	0.54
velocity	9.14%	0.06%	98.75%	0.52
whirr	16.79%	0.13%	97.81%	0.56

5.2.2 RQ2: How Accurately Can Co-Changes Be Predicted Using Structural Dependencies

To answer this research question, we relied on the dependencies dataset (Table 5.1) to build a classification model: the number of dependencies of each kind is taken as a feature, and the occurrence (or absence) of co-changes is taken as the class under observation. Both in RQ2 and in RQ3, we used a reduced sample of 10 projects.

While preparing the dataset, we did not find correlated variables nor redundant variables for any of the systems (Section 5.1.3). This indicates that all variables are sending a distinct signal to the classification model. Table 5.7 shows some characteristics of the dependencies dataset.

The number of rows that had at least one dependency in the dataset varies substantially from project to project, going from 3.20% (mahout) to 42.89% (*velocity*). In all projects, except for tentacles (the smallest project), the percentage of rows indicating *no co-change* was substantially higher than the percentage of rows indicating *co-change*.

Once we filtered out the rows with no dependencies (i.e., rows where all features were not present), we ran the Random Forests algorithm and calculated the AUC of each classifier. The results are depicted in Table 5.8.

Results indicate that, most of the time, the classifier goes for the most frequent class and guesses *no co-change*. By doing so, it achieves low error rates for this class most of the times. However, this

Table 5.9: *Variable Importance: Mean Decrease in Accuracy*

	bval		commons		derby		mahout		scout		tentacles		tomcat		tomee		velocity		whirr	
	value	rank	value	rank	value	rank	value	rank	value	rank	value	rank	value	rank	value	rank	value	rank	value	rank
#TypeImport	0.0056	2	0.0133	4	7.17E-05	3	0.0081	2	0.0084	4	0.0162	5	0.0014	2	7.00E-4	1	0.0014	2	0.0243	1
#MethodImport	0.0001	9	0.0004	11	0.00E+00	5	0	11	0	12	0	7	0	9	00.E+00	6	0	9	0.002	6
#ClassInheritance	0.0019	6	0.0029	9	-8.96E-06	8	0.0005	9	0.0025	7	0	7	0.0002	6	0.00E+00	6	0	9	0.0018	7
#InterfaceImpl	0.0043	3	0.0095	5	1.46E-04	1	0.0027	4	0.0021	9	0	7	0.0003	4	0.00E+00	6	0.0007	4	0.0001	10
#Reference	0.0016	7	0.0144	3	1.22E-04	2	0.0028	3	0.0114	3	0.0233	4	0.0003	4	2.00E-04	4	0.0009	3	0.0078	4
#ReturnType	0.0013	8	0.0055	6	-1.50E-05	9	0.0007	8	0.0036	6	0.0023	6	0	9	0.00E+00	6	0.0001	8	0.0015	8
#Parameter	0.0022	5	0.0029	9	-2.10E-05	10	0.0009	7	0.0074	5	0.0361	2	0.0002	6	1.00E-04	5	0.0002	6	0.0029	5
#Throws	0	11	0.0043	8	-4.57E-06	7	0.0001	10	0.0023	8	0	7	0	9	0.00E+00	6	0	9	0	11
#Access	0.0001	9	0.0046	7	-3.47E-05	11	0.001	6	0.0017	10	0	7	0.0002	6	0.00E+00	6	0.0002	6	0.001	9
#MethodCall	0.0034	4	0.0205	2	3.00E-05	4	0.0021	5	0.0286	1	0.0698	1	0.0006	3	4.00E-04	2	0.0006	5	0.0087	3
#InterfaceInheritance	0	11	0	12	0.00E+00	5	0	11	0.0001	11	0	7	0	9	0.00E+00	6	0	9	0	11
Length Indirect Dep.	0.016	1	0.0212	1	-7.47E-05	12	0.0158	1	0.0284	2	0.025	3	0.0028	1	3.00E-04	3	0.0019	1	0.0215	2

behavior naturally yields bad results for the predictions of the rare class (*co-change*). The AUC measure captures this deficiency, as it ranges from 0.52 to 0.76 (the closer to one, the more accurate the model is).

This result supports the idea that the relationship between dependencies and co-changes might not be very straightforward as traditional literature suggests. In addition, the hypothesis that changes propagate through indirect dependencies seems plausible, but calls for further exploration. From a practical perspective, the variable importance assessment shows that type import dependencies is a good candidate to be used in practice, as they are easy to be determined (as compared to indirect dependencies, for example).

The classifiers we built based on dependency information are often inaccurate, with AUC values ranging from 0.52 to 0.76. This implies dependencies are bad predictors for co-changes.

5.2.3 RQ3: What Kinds of Structural Dependencies Best Explain the Occurrence of Co-Changes?

Table 5.9 presents the importance of all variables for all projects. The higher the value, the higher the importance. For each system, we ranked the variables according to their importance. We used shading to highlight the top 3 variables for each project. Even though the winning variables changed from project to project, some of them appeared more often in the top 3 groups, namely: length of indirect dependency (9 out of 10), number of type imports (7 out of 10), number of method call dependencies (6 out of 10), and number of reference dependencies (5 out of 10). The actual scores of variables (including the most important ones) show that dependencies were indeed bad predictors for co-change.

The kinds of dependency that best explain co-change are: length of indirect dependency, number of type imports, number of method calls, and number of references.

5.3 Discussion

Traditional literature in software engineering, early research studies, and practitioners have long conveyed the idea that structural dependencies are paths through which dependencies propagate. However, more recent research has shown that, even though dependencies are somewhat connected to co-changes, it seems that many co-changes are not motivated or linked back to structural dependencies. Starting in 2004, several researchers published studies indicating that co-changes could be much more accurately predicted using historical information (e.g., evolutionary coupling) instead of structural information (Hassan and Holt, 2004; Ying et al., 2004; Zimmermann et al., 2005). Some

years later, researchers started developing more sophisticated change propagation prediction models that relied on several sources of information (e.g., structural dependencies, code ownership, evolutionary coupling) and were adaptive (Hassan and Holt, 2006; Malik and Hassan, 2008). More recent change prediction studies have achieved higher levels of predictive power by using hybrid methods that combine information retrieval (text similarity), dynamic analysis (execution traces), and evolutionary coupling (Dit et al., 2014; Gethers et al., 2012). We highlight, conversely, that when developing a new module (or a new software system altogether), no historical information might be available (if any) and few execution traces might exist, so dependencies would still be a precious source of information.

5.4 Threats to Validity

In this section, we discuss the threats to the validity of our empirical study.

Construct Validity. This kind of validity refers to whether the operational definition of a variable actually reflects the true theoretical meaning of a concept. As in several mining studies (Bavota et al., 2013; D'Ambros et al., 2009; Hassan and Holt, 2004; Zimmermann et al., 2005), we assume that the commit's change-set represents co-changes. In other words, we take the change-set as our operational definition of co-changes. This seems adequate in the particular context of change propagation, since we do not expect developers to perform incomplete changes very often, i.e., address change propagation in more than one commit. In studies where change-sets would be deemed too fine-grained, authors inferred co-changes from the set of files that had to be changed in order to address a certain specific issue registered in an Issue Tracking Systems (Mcintosh et al., 2014; Wiese et al., 2015). In our study, we also applied a filter to ignore commits with crosscutting changes. This filter might accidentally remove large commits that refactor a large portion of the code or that implements a new big feature. Finally, it is impossible to guarantee that JDX indeed captures all existing structural dependencies from a certain codebase, especially given that developers can write code and use language constructs in unexpected and creative ways. However, we have tested JDX to be confident that it works as expected.

Internal Validity. This kind of validity reflects the extent to which a certain study warrants a causal conclusion. Our research design does not attempt to establish a causal relationship between structural dependencies and co-changes. Instead, we only investigate whether their occurrence is associated or independent. In fact, if we were to establish a causal relationship, we would need to guarantee the absence of plausible alternative explanations for the observed covariation of the variables (nonspuriousness) (Shadish et al., 2001). In our context, a number of other variables would possibly lead to alternative explanations for the effects found. For instance, certain artifacts might co-change because they are semantically related (a.k.a., conceptual coupling (Poshyvanyk and Marcus, 2006)), and not because they are structurally connected. More than that, literature studies in change coupling show that non-structurally-related files might co-change frequently (Zimmermann et al., 2005). Indeed, our classification models showed that structural dependencies were bad predictors for co-changes, which corroborates results found in the literature (Hassan and Holt, 2004; Malik and Hassan, 2008).

External Validity. Even though we picked a random sample of projects to analyze, our generalizability power is limited due to the small size of this sample. More precisely, selecting 10 projects

from a population of 165 allowed us to generalize the results with a sampling error of 31% and with a confidence of 95%. In order to achieve a sampling error of 10% with a confidence level of 95%, it would be necessary to evaluate 61 projects. Although we do not have statistical support to make claims beyond SVN projects in ASF, we believe other open source projects would yield similar results due to the heterogeneity of the sample we analyzed in this study. Improving on this aspect is part of our future work.

Limitations. Our toolset has some limitations: it only works for Java. Removed classes and methods are not considered in our analyses, test classes are also ignored, and we do not keep track of class/method renaming. We also highlight that this study focuses on structural dependencies only. Hence, JDX does not infer dependencies from reflection code blocks (Oracle, 2016), dependency injection configuration files (e.g., from Spring, Java EE), or annotations.

5.5 Summary

Since the notorious software crisis acknowledged in the late 60's (Naur and Randell, 1969), practitioners and researchers have sought better ways to develop software. During this quest, some fundamental Software Engineering principles have emerged, including "low coupling". At the same time, developers are frequently challenged to evolve very large codebases, in which several classes depend on one another in many different ways through many different kinds of dependencies. From a change propagation perspective, should developers care about all of these dependencies? Are there specific kinds of dependencies that are more associated with co-changes?

Building on our previous studies (Oliva and Gerosa, 2011, 2015; Oliva et al., 2011) and in recent literature (Geipel and Schweitzer, 2012), we set out to empirically investigate the interplay between structural dependencies and change propagation (co-changes). With the support of a fairly sophisticated toolset, we analyzed a random sample of 10 Java open source systems from the Apache Software Foundation, which were all hosted in SVN. At every commit, we recorded the occurrence of co-changes and dependencies. We extracted co-changes by analyzing the logs of the version control system. In turn, we extract dependencies from the code snapshot corresponding to every commit in the projects' history.

Our results indicated that structurally dependent files are more likely to co-change than independent ones. Our results also indicated that structural dependencies were bad predictors for co-changes (AUC ranging from 0.52 to 0.76), thus supporting the claim that many co-changes occur because of other reasons. The length of indirect dependencies was the best predictor, which was not used by recent studies that attempted to predict co-changes using structural information (among other sources of information).

Chapter 6

Conclusion

Changeability is an essential property of software systems (Brooks, 1987). In this thesis, we set out to investigate the link between structural dependencies and software changes. We analyzed the history of 45 Java projects randomly sampled from the Apache Software Foundation. Our key findings were:

- When a class A depends on another class B and B changes, the likelihood that A will change together with B is 32% in average, being around 20% higher in average than the likelihood found in the case where A does not depend on B. Hence, the "low coupling" principle holds.
- In the majority of cases, our classification models showed that no two kinds of dependencies are redundant, meaning that there is no silver bullet: all kinds of structural dependencies contribute to explaining co-changes. This requires a fairly sophisticated dependency extraction tool.
- Despite the preprocessing and powerful classification algorithm, our classifiers were often inaccurate, implying that structural dependencies are bad predictors for co-changes. In other orders, it is very likely that most co-changes occur because of factors that are not directly associated with structural dependencies.

6.1 Future Work

In this section, we list open avenues of research that can leverage the toolset and data produced in this thesis.

6.1.1 Structural Anti-Patterns

The structural anti-patterns shown in Section 3.1 are natural candidates to be evaluated as a continuation of this work. They point to specific dependency structures that are claimed to cause large ripple effects and maintenance problems. A future work could leverage the tools and all the dependency graphs already built for all the subject systems to investigate whether *breakables* are indeed breakables (and to what extent), whether *butterflies* cause shotgun surgeries (and how frequently), and whether tangles correlate with change coupling relationships. If the claims are true, then IDE would better implement mechanisms to detect these anti-patterns and warn developers about them, especially newcomers.

During the evolution of a software system, some specific pairs or set of elements keep changing together. We say that these elements are change coupled. It would be helpful to discover whether specific dependency structures or anti-patterns (e.g., highly coupled elements) are associated with change coupling. Such elements would be natural candidates for refactoring.

6.1.2 From Dependencies to Structural Relationships

In this thesis, we strictly investigate structural dependencies. A natural extension of this work would be consider more general "structural relationships", such as classes that are in the same level on a inheritance hierarchy, classes that are located in the same module, etc. Investigating these other structural relationships might shed more light into the relationship between architecture and software changes.

Bibliography

Callo Arias, Trosky B., Pieter Spek, and Paris Avgeriou. A practice-driven systematic review of dependency analysis solutions. *Empirical Software Engineering*, 16:544–586, October 2011. ISSN 1382-3256. doi: http://dx.doi.org/10.1007/s10664-011-9158-8. URL http://dx.doi.org/10.1007/s10664-011-9158-8. 11

Thomas Ball, Jung-Min Kim Adam, A. Porter Harvey, and P. Siy. If your version control system could talk... In *ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*, March 1997. 8

Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. An empirical study on the developers' perception of software coupling. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 692–701, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486879. 55

BlackDuck. Compare Languages - Open Hub. https://www.openhub.net/languages/compare. Accessed: 2016-04-01, 2016a. 40

BlackDuck. Black Duck Open Hub Blog - Factoid List. https://www.openhub.net/languages/compare. Accessed: 2016-04-01, 2016b. 47

Grady Booch. *Object Solutions: Managing the Object-Oriented Project.* Addison-Wesley Professional, first edition, 1995.

Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001. ISSN 0885-6125. doi: 10.1023/A:1010933404324. URL http://dx.doi.org/10.1023/A:1010933404324. 46

Leo Breiman and Adele Cutler. *Breiman and Cutler's Random Forests for Classification and Regression*, 2015. https://cran.r-project.org/web/packages/randomForest/randomForest.pdf. Accessed: 2016-04-01. 46

Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Softw. Eng.*, 25(1):91–121, January 1999. ISSN 0098-5589. doi: 10.1109/32.748920. URL http://dx.doi.org/10.1109/32.748920. 15

Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, and Danny Dig. How do centralized and distributed version control systems impact software changes? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 322–333, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568322. URL http://doi.acm.org/10.1145/2568225.2568322. 40

Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987. ISSN 0018-9162. doi: 10.1109/MC.1987.1663532. URL http://dx.doi.org/10.1109/MC.1987.1663532. 14, 57

Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

Larry L. Constantine. Segmentation and design strategies for modular programming. In Barnett and Constantine, editors, *Modular Programming: Proceedings of a National Symposium*. Information & Systems Press, Cambridge, Massachusetts, 1968. 1

Marco D'Ambros, Michele Lanza, and Mircea Lungu. Visualizing co-change information with the evolution radar. *IEEE Trans. Software Eng*, 35(5):720–735, 2009. URL http://doi.ieeecomputersociety.org/10.1109/TSE.2009.17. 8, 16, 55

E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1): 269–271, 1959. ISSN 0029-599X. doi: 10.1007/BF01386390. URL http://dx.doi.org/10.1007/BF01386390.

Bogdan Dit, Michael Wagner, Shasha Wen, Weilin Wang, Mario Linares-Vásquez, Denys Poshyvanyk, and Huzefa Kagdi. Impactminer: A tool for change impact analysis. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 540–543, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2768-8. doi: 10.1145/2591062.2591064. URL http://doi.acm.org/10.1145/2591062.2591064. 55

Beat Fluri, Harald C. Gall, and Martin Pinzger. Fine-grained analysis of change couplings. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '05, pages 66–74, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2292-0. doi: 10.1109/SCAM.2005.14. URL http://dx.doi.org/10.1109/SCAM.2005.14. 13

Brian Foote and Joseph W. Yoder. *Pattern Languages of Program Design*, volume 4. Addison-Wesley Professional, 1999. 16

Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, jun 1999. With contributions by Kent Beck, John Brant, Willima Opdyke, and Don Roberts. 16

Martin Fowler. Reducing coupling. *IEEE Software*, 18(4):102–104, 2001. URL http://www.computer.org:80/software/so2001/s4102abs.htm. 1, 47

Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 172–181, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597075. URL http://doi.acm.org/10.1145/2597073.2597075. 46

Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 13–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1903-2. URL http://dl.acm.org/citation.cfm?id=942803.943741. 16

Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, first edition, 1994. 1

M.M. Geipel and F. Schweitzer. The link between dependency and cochange: Empirical evidence. *Software Engineering, IEEE Transactions on*, 38(6):1432–1444, Nov 2012. ISSN 0098-5589. doi: 10.1109/TSE.2011.91. 1, 13, 20, 35, 36, 56

Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. Integrated impact analysis for managing software changes. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 430–440, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337274. 55

Tudor Gîrba, Stéphane Ducasse, Adrian Kuhn, Radu Marinescu, and Raţiu Daniel. Using concept analysis to detect co-change patterns. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, IWPSE '07, pages 83–89, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-722-3. doi: 10.1145/1294948.1294970. URL http://doi.acm.org/10.1145/1294948.1294970. 16

Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 345–355, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568260. URL http://doi.acm.org/10.1145/2568225.2568260. 46

Frank E. Harrell Jr. *Regression Modeling Strategies: : With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis.* Springer International Publishing, 2nd edition, 2015. ISBN 3319194240. URL http://www.springer.com/it/book/9783319194240. 45

Frank E. Harrell Jr. *Hmisc Package Manual v3.17-1: Redun function.*, 2016a. https://cran.r-project.org/web/packages/Hmisc/Hmisc.pdf (page 232). Accessed: 2016-04-01. 45

Frank E. Harrell Jr. *Hmisc Package Manual v3.17-1: Varclus function.*, 2016b. https://cran.r-project.org/web/packages/Hmisc/Hmisc.pdf (page 368). Accessed: 2016-04-01. 45

Ahmed E. Hassan and Richard C. Holt. Predicting change propagation in software systems. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 284–293, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2213-0. URL http://dl.acm.org/citation.cfm?id=1018431.1021436. 1, 2, 13, 54, 55

Ahmed E. Hassan and Richard C. Holt. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Softw. Engg.*, 11(3):335–367, September 2006. ISSN 1382-3256. doi: 10.1007/s10664-006-9006-4. URL http://dx.doi.org/10.1007/s10664-006-9006-4. 55

Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley Professional, first edition, 1992. 14

Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning with Applications in R.* Springer New York, 2013. ISBN 1461471370, 9781461471370. doi: 10.1007/978-1-4614-7138-7. URL http://dx.doi.org/10.1007/978-1-4614-7138-7. 45, 46

H. Kagdi, M. Gethers, D. Poshyvanyk, and M.L. Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 119 –128, oct. 2010. doi: 10.1109/WCRE.2010.21. 15

Andrew Koenig. Patterns and antipatterns. *Journal of Object-Oriented Programming (JOOP)*, 8 (1):46–48, 1995. 13

Alyson La. Language Trends on GitHub. https://github.com/blog/2047-language-trends-on-github. Accessed: 2016-04-01, 2015. 40

Michele Lanza and Radu Marinescu. *Object-oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.* Springer, first edition, 2006. xi, 1, 16, 17

Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Prentice Hall, third edition, 2004. 1, 47

Manny Lehman, Dewayne Perry, Juan Ramil, Wladyslaw Turski, and Paul Wernick. Metrics and laws of software evolution–the nineties view. In *Proceedings IEEE International Software Metrics Symposium (METRICS'97)*, pages 20–32, Los Alamitos CA, 1997. IEEE Computer Society Press. doi: 10.1109/METRIC.1997.637156. 14, 47

Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Softw. Eng.*, 34(4):485–496, July 2008. ISSN 0098-5589. doi: 10.1109/TSE.2008.35. URL http://dx.doi.org/10.1109/TSE.2008.35. 46

H. Malik and A.E. Hassan. Supporting software evolution using adaptive change propagation heuristics. In *IEEE International Conference on Software Maintenance, 2008. ICSM 2008.*, pages 177–186, Sept 2008. doi: 10.1109/ICSM.2008.4658066. 13, 55

Cristina Marinescu, Radu Marinescu, Petru Florin Mihancea, Daniel Ratiu, and Richard Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005) - Industrial and Tool volume*, pages 77–80, 2005. ISBN 9-6346-0980-5. 16

Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2213-0. URL http://dl.acm.org/citation.cfm?id=1018431.1021443. 16

Robert Martin. Design principles and design patterns. Technical report, Object Mentor, 2000. URL https://sites.google.com/site/unclebobconsultingllc/home/articles. 15

Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C#.* Prentice Hall, first edition, 2006. 1, 15, 47

S. Mcintosh, B. Adams, M. Nagappan, and A. E. Hassan. Mining co-change information to understand when build changes are necessary. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 241–250, Sept 2014. doi: 10.1109/ICSME.2014.46. 55

Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, pages 1–44, 2015. ISSN 1573-7616. doi: 10.1007/s10664-015-9381-9. URL http://dx.doi.org/10.1007/s10664-015-9381-9. 45

Hayden Melton and Ewan D. Tempero. An empirical study of cycles among classes in java. *Empirical Software Engineering*, 12(4):389–415, 2007. URL http://dx.doi.org/10.1007/s10664-006-9033-1.

Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.*, 27(4):364–380, April 2001. ISSN 0098-5589. doi: 10.1109/32.917525. URL http://dx.doi.org/10.1109/32.917525. 17

Peter Naur and Brian Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968.* NATO Science Committee, 1969. URL http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF. 1, 36, 56

Gustavo Ansaldi Oliva. JDX GitHub Page. https://github.com/golivax/JDX. Accessed: 2016-04-01, 2016a.

Gustavo Ansaldi Oliva. XFlow GitHub Page. https://github.com/golivax/xflow2. Accessed: 2016-04-01, 2016b.

Gustavo Ansaldi Oliva and Marco Aurelio Gerosa. On the interplay between structural and logical dependencies in open-source software. In *Proceedings of the 2011 25th Brazilian Symposium on Software Engineering*, SBES '11, pages 144–153, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4603-2. doi: http://dx.doi.org/10.1109/SBES.2011.39. URL http://dx.doi.org/10.1109/SBES.2011.39. 21, 56

Gustavo Ansaldi Oliva and Marco Aurélio Gerosa. Experience report: How do structural dependencies influence change propagation? an empirical study. In *Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering*, ISSRE 2015, pages 250–260. IEEE, 2015. ISBN 978-1-5090-0405-8. doi: http://dx.doi.org/10.1109/ISSRE.2015.7381818. 39, 56

Gustavo Ansaldi Oliva, Francisco W.S. Santana, Marco A. Gerosa, and Cleidson R.B. de Souza. Towards a classification of logical dependencies origins: a case study. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 31–40, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0848-9. doi: http://doi.acm.org/10.1145/2024445.2024452. URL http://doi.acm.org/10.1145/2024445.2024452. 56

Joshua O'Madadhain. GitHub - jrtom/jung: JUNG: Java Universal Network/Graph Framework. https://github.com/jrtom/jung. Accessed: 2016-05-01, 2016.

Oracle. *Trail: The Reflection API (The Java Tutorials)*, 2016. https://docs.oracle.com/javase/tutorial/reflect. Accessed: 2016-04-01. 56

Meilir Page-Jones. Comparing techniques by means of encapsulation and connascence. *Commun. ACM*, 35(9):147–151, September 1992. ISSN 0001-0782. doi: 10.1145/130994.131004. URL http://doi.acm.org/10.1145/130994.131004. 7, 20

Meilir Page-Jones. *Fundamentals of Object-Oriented Design in UML*. Addison-Wesley, first edition, 1999. 7

David Lorge Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-5855-X. URL http://dl.acm.org/citation.cfm?id=257734.257788. 15

Denys Poshyvanyk and Andrian Marcus. The conceptual coupling metrics for object-oriented systems. In *Proceedings of the 22Nd IEEE International Conference on Software Maintenance*, ICSM '06, pages 469–478, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2354-4. doi: 10.1109/ICSM.2006.67. URL http://dx.doi.org/10.1109/ICSM.2006.67. 20, 27, 55

Roger Pressman. *Software Engineering: A practitioner's approach*. McGraw-Hill, seventh edition, 2009. 1

Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 167–176, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094824. URL http://doi.acm.org/10.1145/1094811.1094824. 1, 17

Francisco Santana, Gustavo Oliva, Cleidson R. B. de Souza, and Marco Aurélio Gerosa. Xflow: An extensible tool for empirical analysis of software systems evolution. In *Proceedings of the VIII Experimental Software Engineering Latin American Workshop*, ESELAW '11, 2011. 27, 41

William Shadish, Thomas Cook, and Donald Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Wadsworth Publishing, 2nd edition, 2001. ISBN 0395615569. 55

Igor Steinmacher, Tayana Conte, Christoph Treude, and Marco A. Gerosa. Overcoming open source project entry barriers with a portal for newcomers. In *38th International Conference on Software Engineering, 2016*, ICSE2016, pages 1–12, New York, NY, USA, 2016. ACM. 2

Wayne P. Stevens, Glenford James Myers, and Larry L. Constantine. Structured design. *IBM Syst. J.*, 13(2):115–139, June 1974. ISSN 0018-8670. doi: 10.1147/sj.132.0115. URL http://dx.doi.org/10.1147/sj.132.0115. 1

Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, Akinori Ihara, and Kenichi Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 812–823, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818852. 45, 46

Igor Scaliante Wiese, Rodrigo Takashi Kuroda, Reginaldo Re, Gustavo Ansaldi Oliva, and Marco Aurélio Gerosa. An empirical study of the relation between strong change coupling and defects using history and social metrics in the apache aries project. In Ernesto Damiani, Fulvio Frati, Dirk Riehle, and Anthony I. Wasserman, editors, *Open Source Systems: Adoption and Impact*, volume 451 of *IFIP Advances in Information and Communication Technology*, pages 3–12. Springer International Publishing, 2015. ISBN 978-3-319-17836-3. doi: 10.1007/978-3-319-17837-0_1. URL http://dx.doi.org/10.1007/978-3-319-17837-0_1. 55

Wikipedia. Coupling (computer programming). https://en.wikipedia.org/wiki/Coupling_(computer_programming). Accessed: 2016-04-01, 2016. 1

F. G. Wilkie and B. A. Kitchenham. Coupling measures and change ripples in c++ application software. *J. Syst. Softw.*, 52(2-3):157–164, June 2000. ISSN 0164-1212. doi: 10.1016/S0164-1212(99)00142-9. URL http://dx.doi.org/10.1016/S0164-1212(99)00142-9. 15

Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, April 1971. ISSN 0001-0782. doi: 10.1145/362575.362577. URL http://doi.acm.org/10.1145/362575.362577. 1

Stephen S. Yau, J. S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *The IEEE Computer Society's Second International Computer Software and Applications Conference*, pages 60–65. IEEE Press, November 1978. 15

Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30:574–586, September 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.52. URL http://dl.acm.org/citation.cfm?id=1018037.1018388. 54

Carlo Zapponi. GitHut - Programming Languages and GitHub. http://githut.info. Accessed: 2016-04-01, 2014. 40

T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 73 – 83, sept. 2003. doi: 10.1109/IWPSE.2003.1231213. 35

Thomas Zimmermann and Peter Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings 1st International Workshop on Mining Software Repositories (MSR 2004)*, pages 2–6, Los Alamitos CA, 2004. IEEE Computer Society Press. 13

Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, 31:429–445, June 2005. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2005.72. URL http://dx.doi.org/10.1109/TSE.2005.72. 2, 54, 55